# Cuprins

**CUPRINS.....1**

**INTRODUCERE.....2**

**CE ŞANSE AM SĂ DEVIN UN BUN PROGRAMATOR ?.....3**

**LEGILE SUCCESULUI DURABIL (GHIDUL STUDENTULUI ÎNDĂRĂTNIC).....6**

**PROBLEME DE JUDECATĂ.....8**

Probleme de perspicacitate.....8
Probleme cu chibrituri.....9
Probleme de logică şi judecată.....10
Probleme de logică şi judecată cu "tentă informatică".....12

**NOŢIUNI FUNDAMENTALE DE PROGRAMARE.....15**

1.Cele trei etape ale rezolvării unei probleme cu ajutorul calculatorului.....15
2.Cum se stabileşte corectitudinea şi eficienţa soluţionării ?.....15
3. Noţiunile fundamentale ale programării: algoritm, limbaje de descriere a algoritmilor, program, limbaje de programare.....16
*3.1. Algoritmul.....16*
*3.2. Descrierea algoritmilor.....17*
*3.3 Programul.....18*
4. Secretul învăţării rapide a programării.....19

**NOŢIUNI PRIMARE DE PROGRAMARE ÎN PASCAL ŞI C.....20**

Exemple de probleme rezolvate.....21

**METODA PRACTICĂ DE ÎNVĂŢARE CE GARANTEAZĂ REZULTATE IMEDIATE.....26**

**PROBLEME SELECŢIONATE - ENUNŢURI.....26**

Probleme propuse spre rezolvare (probleme de antrenament).....26
Probleme de examen.....28
Probleme dificile.....30
Probleme nesoluţionate încă.....32
Probleme insolvabile algoritmic.....35

**NOŢIUNI APROFUNDATE DE PROGRAMARE.....37**

Metode şi strategii de proiectare a algoritmilor (alias tehnici de programare).....37
BackTracking. .....39
Greedy.....42
Programarea dinamică. .....42
Branch & Bound.....43
Recursivitatea.....44

**PROBLEME REZOLVATE ŞI EXERCIŢII DE PROGRAMARE.....49**

Probleme elementare. Exerciţii de programare.....49
Probleme ce necesită back-tracking.....66
Probleme cu soluţie surprinzătoare.....70
Elemente de programare a PC - urilor.....75
Curiozităţi şi trucuri de programare.....91

**CONFRUNTARE DE OPINII: INFORMATICĂ VERSUS MATEMATICĂ.....93**

**BIBLIOGRAFIE, ADRESE ŞI LOCAŢII DE INTERES PE INTERNET.....95**

# Introducere

Există multe culegeri de probleme de informatică ce permit învățarea şi perfecționarea în programare. Prin această culegere am încercat nu doar să sporim această mulțime cu încă una ci să oferim un punct de vedere nou, original şi incitant. Originalitatea nu este dată de enunțurile problemelor sau de rezolvările oferite, ci de ideile şi sfaturile cu caracter mobilizator pe care le oferim, precum şi de faptul că am introdus cîteva capitole cu conținut mai puțin obişnuit într-o culegere de probleme de programare.

Ni s-a părut mai important ca în aceste vremuri, caracterizate prin cuvintele "mă simt într-o permanentă criză de timp", să oferim cît mai mult din experiența noastră directă, atît cea de programator cît şi cea de profesor de programare. Deşi nu credem că există metode perfecte de predare sau de învățare a programării, totuşi sperăm că prin asimilarea informațiilor originale oferite eficiența procesului de învățare a programării în limbajele C şi Pascal va creşte. Este important ca informațiile suplimentare să fie asimilate gradat şi numai în limita "suportabilității" fiecăruia. De aceea, în paginile ce urmează veți găsi şi o serie de informații şi sfaturi ce sintetizează experiența didactică acumulată ca profesor de informatică şi urmîndu-le vă asigurăm că veți obține succesul în programare.

În primele capitole a fost pus un accent important pe motivarea inițială a celor ce doresc să învețe programare. În capitolul "Ce şanse am să devin un bun programator" sînt chiar prezentate cu sinceritate înzestrările necesare unui bun programator.

Tot astfel se explică motivul introducerii unui capitol ce conține probleme de judecată. Rezolvarea acestora pot fi considerate nu doar ca un excelent antrenament al minții ci şi ca o bună ocazie de a aprinde pasiunea pentru informatică şi de a întări motivația programatorilor începători.

Asta nu înseamnă că această culegere nu le este utilă şi celor care au dobîndit deja suficiente cunoştințe de programare. Am introdus în ea cîteva capitole ce conțin informații mai puțin cunoscute. Unul cuprinde o listă de probleme deosebite, unele foarte dificile, altele cărora nu li se cunoaşte încă o soluție şi altele pentru care există demonstrație riguroasă că nu pot fi rezolvate cu ajutorul calculatorului. Alt capitol cuprinde exemple de programare a PC-urilor: lucrul cu tastatura, mouse-ul, accesul direct la memoria ecran, etc. Iar unele capitole ca *Noțiuni aprofundate de programare, Probleme cu soluție surprinzătoare* sau *Curiozități şi trucuri de programare* le sînt în întregime destinate celor care au depăşit stadiul de începător. Probabil că aceste informații constituie o provocare destul de substanțială chiar şi pentru cei avansați în ale programării.

În concluzie, scopul acestei culegeri nu este doar de a contribui la formarea şi specializarea programatorilor sau pentru aprofundarea tehnicilor de programare, cît mai ales de a le oferi o bază, o motivație şi o inițiere celor care doresc să facă primii paşi în domeniul programării. Iar acelor împătimiți ai programării care se simt deja plictisiți, sătui sau plafonați le promitem că parcurgînd această culegere vor aprofunda cunoştințele pe care şi le-au însuşit deja şi, dacă vor avea curajul de "a se lua de piept" cu unele din problemele nesoluționate încă, li se va reaprinde cu siguranță focul pasiunii pentru programare.

Începătorilor le urăm *Bun venit în programare* şi tuturor *Mult succes !*

# Ce şanse am să devin un bun programator ?

Această întrebare apare deseori în discuțiile sincere dintre profesori şi studenții lor descurajați de întîrzierea apariției unor rezultate care să certifice buna lor calitate ca programatori. Vom încerca în rîndurile ce urmează să răspundem cît mai clar la această întrebare oferind, în plus, o perspectivă prospătată asupra acestui subiect, prin luarea în considerare a unei serii de factori mai puțin utilizați în procesul didactic contemporan.

Mai întîi să vedem ce s-ar putea înțelege prin sigtagma ***"bun programator",*** insisitînd în continuare doar pe aprofundarea adjectivului *bun,* fără a mai defini sau detalia ce se înțelege printr-un programator. Vom cita cuvintele recente ale lui Timoty Budd ( profesor la Oregon State University ) care dă următoarea definiție: "*Un bun programator trebuie să fie înzestrat cu tehnică, experiență, capacitate de abstractizare, logică, inteligență, creativitate şi talent*". Întru-totul de acord cu această definiție vom trece în cele ce urmează la explicitarea fiecărei calități.

Înainte vom deduce următoarea consecință imediată - deosebit de importantă - ce rezultă din definiția de mai sus: **cele şapte calități trebuie să fie prezente *toate*** pentru a se obține calificativul de *bun programator.* Deci, prin lipsa sau prin prezența "atrofiată" a uneia , sau a mai multe din "ingredientele rețetei" de mai sus, acest calificativ nu mai poate fi atins.

1. *Tehnica* – este desigur o calitate ce poate fi, şi este, dobîndită doar prin aplicarea asiduă (conform proverbului: "exercițiul îl face pe maestru") în activitatea concretă de programare a tehnicilor de programare învățate şi asimilate de către programator în timpul formării sale profesionale. Nu este exclusă aici posibilitatea obținerii tehnicii de programare înafara unui cadru specializat (într-o facultate de profil ), ci chiar există posibilitatea obținerii ei prin studiu individual şi formație proprie (autodidact ).
2. *Experiența* – este perechea geamănă a calității de mai înainte, fără însă a se exclude una pe cealaltă. Nu vom mai repeta cum şi în ce condiții poate fi ea obținută ci vom deduce următoarea consecința imediată : *nici un programator începător nu poate fi numit* **bun programator** *întrucît el nu a avut cînd (adică timpul necesar ) să dobîndească ambele calități*. Este binecunoscut faptul că o rubrică importantă ce se cere completată la angajare sau la schimbarea locului de muncă este *experiența de programare în ani.* Se consideră în general că experiența apare abia după minimum doi ani de programare. Acest fapt nu trebuie privit ca o descurajare pentru cei mai tineri programatori ci mai degrabă ca pe un motiv de ambiționare şi ca o invitație la rapidă autoperfecționare.
3. *Abstractizarea* – este o trăsătură a intelectului uman şi constituie un dat al oricărui om normal, dar din păcate(!) este o însuşire prea puțin dezvoltată şi prea puțin folosită de oamenii obişnuiți. Ea constă din capacitatea de a extrage din context, de a vedea dincolo de suprafața imediată şi de a putea sesiza structura – scheletul ce susține întreaga rețea de detalii ale unei probleme generale. Pentru a fi un bun programator acestă calitate trebuie să fie net amplificată față de "normal" întrucît stă la baza oricărui proces de analiză şi modelare a problemelor, cît şi la baza procesului de proiectare a soluțiilor generale. Absența sau mai exact atrofierea acestei capacități se constată practic la studenți prin incapacitatea de a înțelege sau de a asimila explicații, demonstrații sau modele abstracte ( simplu spus, o acută şi permanentă "*lipsă de chef*" atunci cînd sînt atinse anumite subiecte ce nu mai au contact direct cu realitatea concretă, imediată – adică subiecte abstracte ). Metoda pentru a recăpăta sau a amplifica această capacitate este de a face cît mai des uz de ea, adică de a o exersa mereu (conform zicalei "*funcția creează organul*") într-un domeniu particular, susținut de o motivație personală puternică. Altfel spus, capacitatea noastră de abstractizare se va amplifica dacă vom încerca găsirea de soluții la problemele dintr-unul din **domeniile noastre preferate**, pentru că rezolvarea acestora va fi automotivată, făcută "cu chef" şi va prezenta o doză sporită de atractivitate.
4. *Logica* – este o altă calitate intrinsecă a oricărui intelect sănătos. Ea este absolut necesară atît pentru a putea folosi mecanismele mentale de deducție şi inducție logică, cît şi pentru a putea înțelege uşor, dar în acelaşi timp corect, cursul – firul roşu al unei demonstrații sau al unui raționament întins pe mai multe pagini. Asemenea tuturor calităților intrinseci existente în stare potențială, antrenarea şi amplificarea acesteia se face prin exercițiu repetat, prin folosirea ei în mod curent.Din păcate, doar prin rezolvarea de integrame nu se ajunge la amplificarea logicii…
5. *Inteligența* – este una din cele mai de preț calități intrinseci ale intelectului uman. În cîteva cuvinte, fără a avea pretenția de a da prin acestea o definiție, prin inteligență înțelegem capacitatea de a face (de a stabili) conexiuni sau legături noi şi folositoare (din latinescul *inter-legere*) între idei,

cunoştinţe sau informaţii “aparent fără legătură”. Faţă de logică, pe care o considerăm ca fiind o calitate bazală, inteligenţa este o calitate ce se “întinde pe verticala” intelectului şi are în plus trăsătura de a fi mult mai dinamică şi mai mobilă (chiar fulgerător de rapidă) în acţiune. Pentru cultivarea, amplificarea şi cizelarea acestei calităţi este nevoie de “punerea ei la lucru” cît mai des şi pe durate tot mai mari de timp. Insatisfacţia obţinerii unor rezultate rapide sau chiar imediate este un obstacol ce poate fi depăşit relativ uşor prin antrenarea inteligenţei pe un “teren” cunoscut şi accesibil, adică în **domeniul preferat de interes**. În acest fel există siguranţa de a fi susţinut de atracţia sporită pentru acel domeniu particular ceea ce va conduce prin efort perseverent (dar susţinut de această dată cu pasiune !) la apariţia rezultatelor aşteptate şi, implicit, a satisfacţiei.

6. *Creativitatea* – este o calitate intrinsecă nu numai intelectului uman ci însăşi vieţii în general. Ea constă, în ultimă instanţă, în capacitatea de a face (de a produce) ceva cu ***adevărat nou şi original***. De aceea am putea afirma că toate organismele vii, prin capacitatea lor de a se opune entropiei, creează mereu ordine din dezordine aducînd în acest fel ceva nou, neaşteptat. Ceea ce se aşteaptă însă de la un bun programator nu este doar acest tip de creativitate (gen: *adaptare inconştientă şi instinctivă*) ci o creativitate conştientă, responsabilă, reflectată în adaptarea soluţiilor existente sau chiar inventarea altora noi. În acest sens trebuie să menţionăm că există o legătură strînsă, dovedită şi verificată în practică (chiar dacă pare oarecum inexplicabil la prima vedere), între *creativitate – inteligenţă fluidă – curiozitate – sublimarea impulsurilor erotice - umor şi poftă de viaţă*. **Cultivarea şi amplificarea controlată** a oricărora dintre aceste patru trăsături va conduce în mod automat la amplificarea şi dinamizarea creativităţii intelectuale.
7. *Talentul* – este singura calitate ce nu poate fi cultivată şi amplificată. În accepţiunea sa obişnuită, prin talent se înţelege o sumă de înzestrări native sau o predispoziţie personală pentru un anumit domeniu. Existenţa talentului este percepută de cel în cauză ca ***uşurinţă – abilitate - dexteritate*** de a învăţa, asimila şi aplica toate cunoştinţele domeniului respectiv, abilitate ce este simţită de cel "talentat" ca un fel de *“ceva în plus”* în comparaţie cu capacităţile celor din jur. Din păcate, în accepţiunea comună se crede că talentul este calitatea suficientă care permite oricui atingerea cu siguranţă a calificativului *bun programator*, concepţie este infirmată de orice programator cu experienţă. Asta nu înseamnă că lipsa talentului în programare este permisă pentru atingerea acestui nivel, însă efortul, tenacitatea şi răbdarea existente în “cantităţi” mult sporite într-o asemenea situaţie de ne-înzestrare cu talent vor permite o apropiere sigură de acest calificativ. Din păcate, lipsa talentului va apărea la început sub forma unei insatisfacţii interioare şi ca o impresie acută că lipsesc rezultatele. Reamintim că însăşi cuvîntul *facultate* are la origine sensul de *capacitate, potenţialitate, înzestrare.* Deci, normal ar fi ca alegerea unui student pentru frecventarea cursurilor unei Facultăţi să fi fost făcută ţinînd cont de aptitudinile şi abilităţile celui în cauză, descoperite în prelabil, adică să se dovedească talentat pentru domeniul ales. Acest lucru este cu atît mai important în cazul optării pentru învăţarea *programării,* cunoscută fiind ca o specializare complexă şi solicitantă.

Reluînd în sinteză ideile prezentate, putem spune că:

- Pentru a fi un *bun programator* trebuie să fie prezente următorele şapte calităţi într-o formă activă, dinamică: *tehnică, experienţă, capacitate de abstractizare, logică, inteligenţă, creativitate şi talent*.
- Dintre toate cele şapte calităţi necesare programării de înaltă calitate, numai una – *talentul* - nu este inerentă unui intelect sănătos. De altfel, prezenţa talentului nu este absolut necesară pentru a deveni programator, dar în timp ce absenţa lui îngreunează apropierea de calificativul *bun programator,* prezenţa lui şi amplificarea celorlalte calităţi este o garanţie a succesului, ce va fi cu siguranţă obţinut, însă nu fără efort, răbdare şi perseverenţă !
- Toate celelalte şase calităţi excluzînd talentul, *prezente fiind într-o formă potenţială*, trebuiesc doar *cultivate şi amplificate*. Am prezentat mai sus în detaliu modul de amplificare a fiecăreia.
- *“Cheia secretă“* ce conduce *cu siguranţă* la declanşarea procesului de dinamizare şi amplificare a fiecăreia din cele şase calităţi inerente este de *a avea mereu o motivaţie puternică* (de a învăţa “cu chef” sau “cu tragere de inimă” !). Acest fapt este posibil dacă se ţine cont de necesitatea adaptării efortului la domeniul preferat al celui în cauză. La modul concret, este necesar ca toate aplicaţiile, problemele, exerciţiile, întrebările, curiozităţile, inovaţiile, descoperirile, “săpăturile”, etc., să fie făcute sau să fie alese, la început, din domeniul preferat (hobby-ul), chiar dacă acesta nu are la prima vedere legătură cu programarea. Scopul ce se atinge cu siguranţă în acest mod în această primă fază este acela de a pune “la lucru” inteligenţa, creativitatea, logica, etc., ceea ce va conduce cu siguranţă la trezirea şi amplificarea rapidă a acestor calităţi. Acest fapt va permite apoi trecerea la o a doua

fază în care, pe baza acumulărilor calitative obținute, se poate trece la programarea propriu-zise "înarmat cu forțe proaspete".

Încheiem răspunzînd într-o singură frază întrebării din titlu *Ce şanse am să devin un bun programator ?* :

**dacă mă simt înzestrat cu *talent* pentru programare (adică nu mă simt inconfortabil la acest subiect) atunci, mobilizîndu-mi voința (motivația) şi amplificîndu-mi *capacitatea de abstractizare, logica, inteligența şi creativitatea (*ce există în mine într-o formă potențială), prin practică de programare voi acumula în timp *tehnica* şi *experiența* necesare pentru a deveni cu siguranță un *bun programator* , însă nu fără efort, răbdare şi perseverență.**

## Legile succesului durabil (Ghidul studentului îndărătnic)

Cunoaşte-ţi **Regulile de aur ale studentului şmecher** ? Dacă nu, le puteţi fi afla “la o bere”, de la şmecher la şmecher. Noi le vom numi **"Anti-legile succesului durabil"** şi vi le prezentăm în continuare doar pentru a putea observa cum fiecare din aceste "legi" este o răsturnare (pervertire) a adevăratelor legi ale succesului.

1. Cel mai important este să termini facultatea şi să te vezi cu diploma în mînă. Ce contează cum ? Cine mai ştie dup-aia… ?
2. De ce să-nveţi …? Şi aşa majoritatea materiilor sînt tembele şi n-o să-ţi folosească niciodată în viaţă. …materiile tembele trebuie să fie predate numai pentru ca să cîştige şi profii’ o pîine.
3. Pune-te bine cu profesorii pînă treci examenul. Stai cu ei la o ţigară în pauză. Lasă-i pe ei să vorbească. Tu prefă-te că eşti interesat…
4. Ai trecut examenul ? Da ? Atunci… restul nu mai contează.
5. Nu contează dacă ai învăţat, ce ştii sau cît ştii. Important este să ai baftă la examen, să ai mînă bună sau să mergi "bine pregătit"… La puţini profi’ nu se poate copia !
6. Sînt examene la care, se ştie bine, toată lumea copiază. Trebuie să fi nebun să-nveţi la ele !
7. Notele bune sînt numai pentru piloşi şi tocilari.

Acestor studenţi le sînt însă complet necunoscute **Legile succesului durabil.** Ele ar putea fi intuite doar de acei puţini care s-au format şi s-au educat în spiritul ideilor ce urmează să le explicăm în continuare. Aceste legi ne învaţă că bazele succesului durabil se pun încă din timpul şcolii şi mai ales din timpul facultăţii. Şi ne mai învaţă că succesul astfel "start-at" este destinat să dureze o viaţă întreagă.

1. ***Cel mai important în facultate este să-ţi faci o carte de vizită***, nu-i suficient să “vînezi” doar diploma. Dacă vei fi apreciat şi vei ajunge să fii considerat capabil sau chiar bun de cadrele didactice "cu greutate", vei ajunge să fi cunoscut şi bine cotat după absolvire şi-ţi vei găsi un loc bun de muncă. **Întotdeauna a fost şi va fi nevoie de oameni capabili** "pe piaţa muncii", nu de licenţiaţi "piloşi", “tolomaci” sau “papagali”.
2. ***Cel mai important lucru în şcoală este că înveţi cum să înveţi***. Cînd vrei să te recreezi rezolvînd integrame nu prea contează ce din ce domeniu ţi le-ai ales. Important pentru tine nu este cum, ci faptul că te destinzi. Tot astfel, în facultate important este nu neapărat ce înveţi, **ci că înveţi**! Multe cunoştinţe le vei uita în primii ani după absolvire, mai ales cele pe care ţi le-ai însuşit într-o stare de sforţare şi încrîncenare, fără plăcere. **Cel mai important este să înveţi de plăcere** căci numai aşa vei învăţa cum să înveţi. Iar aceasta nu se mai poate uita! Şi nu vei mai uita nicicînd că ai resursele şi puterea să treci prin forţe proprii examenele cele mai grele.
3. ***Succesul în viaţă se bazează pe relaţii umane echilibrate***. (Acest fapt era cunoscut şi pe vremea regimului partidului comunist român P.C.R. însă datorită imoralităţii generalizate a societăţii el a fost aplicat pe invers: astfel, a apela de P.C.R. însemna atunci să apelezi la Pile, Cunoştinţe şi Relaţii.) Deci, **cel mai important lucru în şcoală este să înveţi arta de a stabilii relaţii umane prietenoase şi de încredere reciprocă**. Ceea ce va conta cel mai mult, peste ani, este că ai stabilit în timpul şcolii multe prietenii durabile şi de încredere care te vor “îmbogăţii” astfel pentru toată viaţa. În plus, nu uita: şi profesorii sînt oameni. Au şi ei nevoie de prieteni.
4. ***Colegii sînt martori şi devin cei mai exigenţi judecători ai trăsăturilor tale de caracter*.** Examenul, indiferent de materie sau disciplină, cu emoţiile şi peripeţiile lui este în sine o lecţie completă. Nu contează atît dacă l-ai luat sau dacă l-ai picat, **ci contează cum**! Contează ce fel de om eşti în astfel de situaţii, cînd tocmai îţi construieşti “cartea de vizită sau blazonul”. Nu uita că nu te afli doar în faţa profesorilor ci eşti tot timpul înconjurat de colegii care te judecă, chiar dacă ţi-e nu-ţi spun. Pentru că **aşa cum te comporţi acum în examen, aşa te vei comporta toată viaţa**.
5. ***Examenele grele sînt cele care îţi pot forma un caracter puternic***. Ceea ce este important în examen, ca şi în situaţiile de viaţă, este încrederea în reuşită şi stăpînirea de sine chiar dacă n-ai învăţat toată materia. **Dacă ai învăţat destul ca să te simţi stăpîn pe tine atunci ai trecut examenul !** Chiar acesta a fost rostul lui, ce dacă ţi-a dat notă mică! Crezi că, după ce vei trece examenul, peste zece ani îţi vei mai aminti cu ce notă ?
6. ***Cei cu un caracter slab şi vicios se vor da la un moment dat în vileag*.** Cei care copiază nu-şi dau seama că ei îşi “infectează” caracterul. Şi nici cît de grave sînt consecinţele infectării cu “microbul” cîştigului imediat obţinut prin furt. Oare se vor mai putea debarasa vreodată de acest viciu tentant ? Dar de cunoscutele "efecte secundare": sentimentul de nesiguranţă fără o fiţuică în buzunar, atracţia irezistibilă pentru “aruncarea privirii” împrejur, părerea de rău că “Ce prost sînt,

puteam să copiez tot !", etc. cînd vor mai scăpa ? **Cei care se obişnuiesc să copieze, atît cît vor trăi, vor fi jumătate om-jumătate fiţuică**. Ca în vechile bancuri cu milițieni…

7. ***Oricine este acum apt să înveţe şi să-şi însuşească pentru întreaga sa viaţă Legea efortului*.** Pe profesori îi impresionează cel mai tare efortul depus şi-l vor aprecia cu note maxime. Ei supra-notează pe cei "care vor, sînt bine intenționați, dar încă nu pot". Profesorii cunosc adevărul exprimat în *Legea omului de geniu (legea lui Einstein)*: "**Geniul este compus 99% din transpirație şi 1% din inspirație**". Profesorii adevărați se străduiesc să noteze mai ales calitatea umană şi profesională a studentului. Rețineți: dacă studentul a fost prietenos, activ şi deschis în timpul anului şcolar şi a depus un efort constant pentru a se perfecționa, fapt ce nu a scăpat ochiului atent al profesorului, examenul devine în final pentru el o formalitate…

Multe vorbe şi păreri pot fi auzite pe această temă în familie, în pauze la şcoală sau la barul preferat. Cît sînt ele de adevărate ? S-ar putea da oare o definiție precisă pentru *succesul în viață* ?

Noi nu cunoaştem o astfel de definiție, ştim doar că există o multitudine de păreri şi opinii, unele profund contradictorii. Este însă de bun simț să credem că se poate numi "de succes" acea viață care este plină de satisfacții, bucurii şi visuri împlinite. Acea viață care să-şi merite din plin exclamația: "Asta da, viață !" ?

***Regula de aur a succesului durabil*** este: **Învață să-ți construieşti singur viața.** Şi apoi, dacă ai învățat, apucă-te fără întîrziere să-ți "faci" viața fericită.

Studenția, prin entuziasmul, optimismul şi idealismul ei, este o perioadă optimă pentru a învăța cum **să-ți faci o viață de succes** ! Atenție, mulți şi-au dat seama prea tîrziu că studenția a fost pentru ei în multe privințe ultimul tren…

## Probleme de judecată

Oferim în cele ce urmează o selecţie de probleme ce nu necesită cunoştinţe de matematică avansate (doar nivelul gimnazial) dar care pun la încercare capacitatea de judecată, inspiraţia şi creativitatea gîndirii. Rezolvarea acestor probleme constituie un bun antrenament pentru creşterea capacităţii de gîndire creativă precum şi a fluidităţii gîndirii. Credem că nu degeaba aceste două trăsături sînt considerate cele mai importante semne ale tinereţii minţii.

Problemele, selectate din multiple surse, nu au putut fi grupate în ordinea dificultăţii mai ales datorită diversităţii şi varietăţii lor. Ele au fost doar separate în cîteva categorii a căror nume vrea să sugereze un anumit mod de gîndire pe care l-am folosit şi noi în rezolvarea lor. Cele cu un grad mai mare de dificultate au fost marcate cu un semn (sau mai multe semne) de exclamare.

Criteriul principal pe baza căruia s-a făcut această selecţie a fost următorul: **fiecare problemă cere în rezolvarea ei un minimum de inventivitate şi creativitate**. Majoritatea problemelor te pun "faţă în faţă cu imposibilul", aşa că rezolvarea fiecărei probleme necesită depăşirea unor "limitări ale gîndirii" plus un minimum de originalitate în gîndire. Tocmai de aceea, pentru rezolvarea lor este nevoie de efort, putere de concentrare şi perseverenţă. Zis într-un singur cuvînt: este necesar şi un strop de **pasiune**.

Considerăm că eforturile consecvente ale celor care vor rezolva aceste probleme vor fi din plin răsplătite prin plăcerea "minţii biruitoare" şi prin amplificarea calităţilor următoare: capacitate sporită de efort intelectual, putere de concentrare mărită şi prospeţime în gîndire.

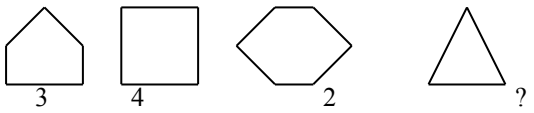
Vă dorim mult succes !

## Probleme de perspicacitate

1. Ştiind că o sticlă cu dop costă 1500 lei şi că o sticlă fără dop costă 1000 lei, cît costă un dop ?

2. Ştiind că un ou costă 1000 lei plus o jumătate de ou, cît costă un ou ?

3. Ce număr lipseşte alături de ultima figură:



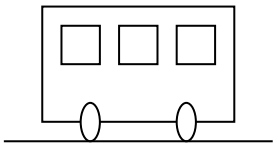

4. Lui Popescu nici prin gînd nu-i trecea să folosească toate mijloacele pe care le avea la îndemînă ca să lupte împotriva adversarilor tendinţei contra neintroducerii mişcării anti-fumat. Care este poziţia lui Popescu: este *pentru* sau *contra* fumatului ?

5. **Împărţirea "imposibilă".** Să se împartă numărul 12 în două părţi astfel încît fiecare parte să fie 7.

6. **9 puncte.** Să se secţioneze toate cele 9 mici discuri cu o linie frîntă neîntreruptă (fără a ridica creionul de pe hîrtie) compusă din 4 segmente. (!) Dar din trei segmente, este posibil ?

7. **Trei cutii.** În trei cutii identice sînt închise trei perechi de fructe: fie o pereche de mere, fie o pereche de pere, fie o pereche formată dintr-un măr şi o pară. Pe cele trei cutii sînt lipite trei etichete: "două mere", "două pere" şi, respectiv, "un măr şi o pară". Ştiind că nici una din etichete nu corespunde cu conţinutul cuitei închise pe care se află, să se afle care este *numărul minim* de extrageri a cîte un fruct pentru a se stabili conţinutul fiecărei cutii.

8. În ce direcţie merge autobuzul din desenul alăturat ?



9. **(!) Întrerupătoarele.** Pe peretele alăturat uşei încuiate de la intrarea unei încăperi, se află trei întrerupătoare ce corespund cu cele trei becuri de pe plafonul încăperii în care nu putem intra. Acționînd oricare din întrerupătoare, dunga de lumină care apare pe sub uşă ne asigură că niciunul din cele trei becuri nu este ars. Cum putem afla, fără a pătrunde în încăpere, care întrerupător corespunde cu care bec ?

10. **(!!) Cine mută ultimul cîştigă.** Doi jucători dispun de o masă de joc de formă circulară sau pătrată şi de un număr mare de monezi identice. Ei mută plasînd pe masa de joc în spațiul neocupat, fără suprapunere, cîte o monedă alternativ pînă cînd unul dintre jucători, care pierde în acest caz, nu mai poate plasa nicăieri o monedă. Să se arate că primul jucător are o strategie sigură de cîştig.

11. **(!!!) Iepurele şi robotul-vînător.** Într-o incintă închisă (un gen de arenă) se află un iepuraş şi un robot-vînător înzestrat cu cleşti, mijloc de deplasare, calculator de proces şi "ochi" electronici. Ştiind că viteza de deplasare a robotului-vînător este constantă şi de zeci de ori mai mare decît a iepuraşului, ce şanse mai are iepuraşul de a scăpa ?

12. **Cîntarul defect.** Avînd la dispoziție un cîntar gradat defect care greşeşte constant cu aceeaşi valoare (cantitate necunoscută de grame), putem să cîntărim ceva determinîndu-i corect greutatea ?

13. **Jocul dubleților (inventat de Carroll Lewis).** Ştiind că trecerea de la un cuvînt cu sens la altul cu sens este permisă doar prin modificarea unei singure litere odată (de exemplu: UNU → UNI → ANI → ARI → GRI → GOI → DOI ) se cere: *Dovediți că IARBA este VERDE şi că MAIMUȚA a condus la OMENIRE, faceți din UNU DOI, schimbați ROZ-ul în ALB, puneți ROUGE pe OBRAZ şi faceți să fie VARA FRIG.*

14. **Împăturirea celor 8 pătrate.** Împăturiți inițial în opt o foaie dreptunghiulară după care desfaceți-o şi însemnați fiecare din cele opt zone dreptunghiulare obținute (marcate de pliurile de îndoire) cu o cifră de la 1 la 8. Puteți împături foaia astfel obținută reducînd-o de opt ori (la un singur dreptunghi sau pătrat) astfel încît trecînd cu un ac prin cele opt pliuri suprapuse acesta să le perforeze exact în ordinea 1, 2, 3, …, 8 ? Încercați aceste două configurații:

| | | | |
|---|---|---|---|
| 1 | 8 | 7 | 4 |
| 2 | 3 | 6 | 5 |

| | | | |
|---|---|---|---|
| 1 | 8 | 2 | 7 |
| 4 | 5 | 3 | 6 |

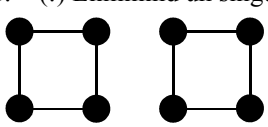
15. **Problemă pentru cei puternici.** Încercați să împăturiți de 8 ori, pur şi simplu, o coală de hîrtie (de fiecare dată linia de îndoire este "în cruce" peste cea dinainte). Este posibil ? (!)Determinați ce dimensiuni ar trebui să aibă foaia la început pentru a putea fi împăturită de 8 ori.

16. Este posibil ca un cal să treacă prin toate cele 64 de pătrățele ale unei table de şah, începînd dintr-un colț şi terminînd în colțul diagonal opus ?

17. Într-un atelier există 10 lădițe ce conțin fiecare piese cu greutatea de 100 grame, cu excepția uneia din lădițe ce conține piese avînd grutatea de 90 grame. Puteți preciza care este lădița cu pricina, folosind un cîntar doar pentru o singură dată ?

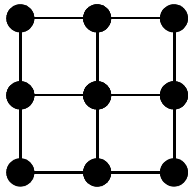
## Probleme cu chibrituri

1. (!) Eliminînd un singur băț de chibrit ceea ce rămîne în fața ochilor este un elipsoid!

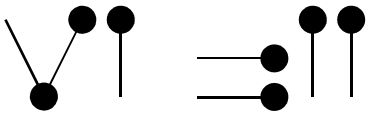


2. **(!) 9 bețe.** Să se aşeze 9 bețe de chibrit astfel încît ele să se întîlnească la vîrf tot cîte trei în şase vîrfuri distincte.

3. **De la 4 la 3.** În figura ce conține 4 pătrate, mutînd 4 bețe să se obțină o figură ce conține doar 3 pătrate.



4. **6 = 2 ?** Mutînd doar un singur băț de chibrit să se restabilească egalitatea:



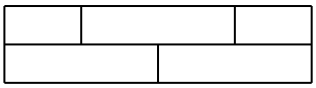
5. **Problema ariilor întregi.** Puteți aşeza 12 chibrituri astfel încît ele să formeze contururile unor poligoane ce au aria întreagă egală cu 5, (!!) 4, 3, 2, (!!!) 1 ? Se subînțelege că un chibrit poate fi asimilat cu un segment de lungime 1 şi că nu există nici o dificultate de a forma "din ochi" unghiuri drepte.

## Probleme de logică şi judecată

1. **Substituirea literelor.** Subtituiți literele cu cifre astfel încît următoarele adunări să fie corecte: GERALD + DONALD = ROBERT ; FORTY + TEN + TEN = SIXTY ; BALON + OVAL = RUGBY.

2. **Test de angajare la Microsoft.** Patru excursionişti ajung pe malul unui rîu pe care doresc să-l traverseze. Întrucît s-a înoptat şi ei dispun doar de o singură lanternă, ei pot să treacă rîul cel mult cîte doi laolaltă. Ştiind că, datorită diferențelor de vîrstă şi datorită oboselii, ei ar avea individual nevoie pentru a traversa rîul de 1, 2, 8 şi 10 minute, se cere să se decidă dacă este posibilă traversarea rîului în aceste conditții în doar 17 minute ?

3. **(!) Imposibilă.** Să se taie toate cele 16 segmente ale figurii următoare cu o singură linie curbă continuă şi care nu se intersectează cu ea însăşi.



4. **(!) Problema "ochilor albaştri".** Sîntem martorii următorului dialog între două persoane X şi Y. << **X:** Eu am trei copii. Produsul vîrstei lor este 36 iar suma vîrstei lor este egală cu numărul de etaje al blocului din vecini de mine. Îl ştii, nu-i aşa ? **Y:** Desigur. Dar numai din cît mi-ai spsus nu pot să deduc care este vîrsta copiilor tăi. **X:** Bine, atunci află că cel mare are ochi albaştrii.>> Puteți afla care este vîrsta celor trei copii ?

5. **Problema călugărului budhist.** Într-o dimineață, exact la răsăritul soarelui, un călugăr budhist porneşte de la templul de la baza muntelui pentru a ajunge la templul din vîrful muntelui exact la apusul soarelui, unde el se roagă toată noaptea. A doua zi el porneşte din vîrf pe aceeşi cărare, tot la răsăritul soarelui, pentru a ajunge la templul de la baza muntelui exact la apusul soarelui. Să se arate că a existat un loc pe traseu în care călugărul s-a aflat în ambele zile exact la aceaşi oră.

6. **Vinul în apă şi apa în vin.** Dintr-o sticlă ce conține un litru de apă este luat un pahar (un decilitru) ce este turnat pest un litru de vin. Vinul cu apa se amestecă bine după care se ia cu acelaşi pahar o

cantitate egală de "vin cu apă" ce se toarnă înapoi peste apa din sticlă. Avem acum mai multă apă în vin decît vin în apă, sau invers ?

7. **(!!!!) Cuiele în echilibru.** Avem la dispoziție 7 cuie normale, cu capul obişnuit. Înfigem unul vertical în podea (sau într-o placă de lemn). Se cere să se aşeze cele 6 cuie rămase în **echilibru stabil** pe capul cuiului vertical, fără ca niciunul din cele şase cuie să atingă podeaua.

8. **(!!) Țigările tangente.** Este posibil să aşezăm pe masă şase țigări astfel încît fiecare să se atingă cu fiecare (oricare două să fie tangente) ? **(!!!)** Dar şapte țigări ?

9. **(!) Problema celor 12 înțelepți (**în variantă modernă). Managerul unei mari companii doreşte să pună la încercare inteligența şi puterea de judecată a celor 12 membrii ai consiliului său de conducere. Luînd 12 cărți de joc, unele de pică şi altele de caro, el le aşează cîte una pe fruntea fiecărui consilier astfel încît fiecare să poată vedea cărțile de pe frunțile celorlalți dar nu şi pe a sa. Managerul le cere celor care consideră că au pe frunte o carte de caro (diamond) să facă un pas în față, altfel ei nu vor mai putea face parte din consiliu. După ce îşi repetă cererea de şapte ori, timp în care niciunul din cei 12 consilieri nu face nici o mişcare (ci doar se privesc unii pe alții), toți consilierii care au într-adevăr pe frunte o carte de caro ies deodată în față. Puteți deduce cîți au ieşit şi cum şi-au dat ei seama ce carte este aşezată pe fruntea lor ?

10. **Păianjenul şi musca.** Pe peretele lateral al unei hale cu dimensiunile de 40 x 12 x12 metri, pe linia mediană a peretelui lateral şi exact la 1 metru de tavan, se află un păianjen. Pe peretele lateral opus, tot pe linia mediană şi exact la 1 metru de podea, se află o muscă amorțită. Care este distanța cea mai scurtă pe care păianjenul o are de parcurs de-a lungul pereților pentru a se înfrupta din muscă ?

11. **Rifi şi Ruf.** Cei doi iubiți Rifi şi Ruf, din nordica țară Ufu-Rufu, locuiesc în sate diferite aflate la distanța de 20 km unul de altul. În fiecare dimineață ei pornesc exact deodată (la răsărit) unul spre celălalt spre a se întîlni şi a se săruta confrom obiceiului nordic: nas în nas. Într-o dimineață o muscă rătăcită porneşte exact la răsăritul soarelui de pe nasul lui Rifi direct spre nasul lui Ruf, care o alungă trimițînd-o din nou spre nasul lui Rifi, ş.a.m.d. ..., pînă cînd ea sfîrşeşte tragic în momentul "sărutului" celor doi. Ştiind că Rifi se deplasează cu 4 km/oră, Ruf cu 6 km/oră iar musca zboară cu 10 km/oră, se cere să se afle ce distanță a parcurs musca în zbor de la răsărit şi pînă în momentul tragicului ei sfîrşit.

12. **O anti-problemă de şah.** În următoarea configurație a pieselor pe o tablă de şah se cere să nu dați mat dintr-o mutare ! (Albul atacă de jos în sus. Legenda: P-pion, N-nebun, R-rege, T-turn, C-cal. Alăturat fiecărei piese este scrisă culoarea sa, alb-a sau negru-n.)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| N Na | | | | | | R Ra | T Ta |
| | T Tn | | | | | | N Na |
| | | | | | | T Ta | |
| | | N Nn | | P Pn | | P Pn | |
| | | P Pa | | R Rn | | P Pa | |
| | P Pn | | | P Pa | | | P Pn |
| | P Pa | | | P Pa | | | P Pa |
| | | | C Ca | | C Ca | | |

13. **Bronx contra Brooklyn.** Un tînăr, ce locuieşte în Manhattan în imediata apropiere a unei stații de metrou, are două prietene, una în Brooklyn şi cealaltă în Bronx. Pentru a o vizita pe cea din Brooklyn el ia metroul ce merge spre partea de jos a oraşului, în timp ce, pentru a o vizita pe cea

din Bronx, el ia din acelaşi loc metroul care merge în direcţie opusă. Metrourile spre Brooklyn şi spre Bronx intră în staţie cu aceeşi frecvenţă: din 10 în 10 minute fiecare. Dar, deşi el coboară în staţia de metrou în fiecare sîmbătă la întîmplare şi ia primul metrou care vine (nedorind să "favorizeze" pe nici una din prietenele sale), el a constatat că, în medie, el merge în Brooklyn de 9 ori din 10. Puteţi găsi o explicaţie logică a fenomenul ?

14. **(!!) Problema celor 12 bile.** În faţa noastră se află 12 bile identice ca formă, vopsite la fel, dar una este cu siguranţă falsă, ea fiind fie mai grea, fie mai uşoară, fiind făcută dintr-un alt material. Avem la dispoziţie o balanţă şi se cere să determinăm doar prin 3 cîntăriri care din cele 12 bile este falsă precizînd şi cum este ea: mai grea sau mai uşoară. **(!!!) Mai mult**, puteţi determina care este numărul maxim de bile din care prin 4 cîntăriri cu balanţa se poate afla exact bila falsă şi cum este ea ?

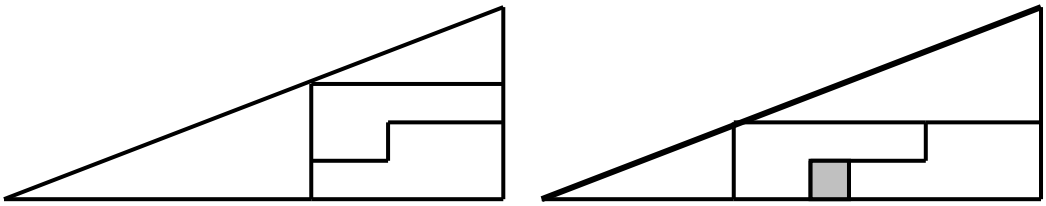
15. **(!) Problema celor 2 perechi de mănuşi.** Aflat într-o situaţie ce implică intervenţia de urgenţă, un medic chirurg constată că are la dispoziţie doar 2 perechi de mănuşi sterile deşi el trebuie să intervină rapid şi să opereze succesiv 3 bolnavi. Este posibil ca cele trei operaţii de urgenţă să se desfăşoare în condiţii de protecţie normale cu numai cele 2 perechi de mănuşi ? (Sîngele fiecăruia din cei 3 pacienţi, precum şi mîna doctorului nu trebuie să conducă la un contact infecţios.)

16. **(!!) Problema frînghiei prea scurte.** O persoană ce are asupra ei doar un briceag şi o frînghie lungă de 30 metri se află pe marginea unei stînci, privind în jos la peretele vertical de 40 metri aflat sub ea. Frînghia poate fi legată doar în vîrf sau la jumătatea peretelui (la o înălţime de 20 metri de sol) unde se află o mică platformă de sprijin. Cum este posibil ca persoana aflată în această situaţie să ajungă teafără jos coborînd numai pe frînghie, fără a fi nevoită să sară deloc punîndu-se astfel în pericol ?

17. **Problema lumînărilor neomogene.** Avem la dispoziţie chibrite şi două lumînări care pot arde exact 60 minute fiecare însă, ele fiind neomogene, nu vor arde cu o viteză constantă. Cum putem măsura precis o durată de 45 minute ?

18. **(!!) O jumătate de litru.** Avem în faţa noastră un vas cilindric cu capacitatea de 1 litru, plin ochi cu apă. Se cere să măsurăm cu ajutorul lui ½ litru de apă, fără a ne ajuta de nimic altceva decît de mîinile noastre.

19. **(!) Să vezi şi să nu crezi.** Priviţi următoarele două figuri: prin reaşezarea decupajelor interioare ale primeia se obţine din nou aceeaşi figură dar avînd un pătrăţel lipsă ! Cum explicaţi "minunea" ?



## Probleme de logică şi judecată cu "tentă informatică"

1. **(!!!) Decriptarea scrierii încifrate.** Se dau următoarele numere împreună cu denumirile lor cifrate:

| | |
|---|---|
| 5 | nabivogedu |
| 6 | nagevogedu |
| 10 | nabivobinaduvogedu |
| 15 | nabivonagevogedunaduvogedu |
| 20 | nabivogenagevogenaduvogedu |
| 25 | nabivonabivobinagevogedunagevogenaduvogedu |
| 30 | nabivodunanabivobiduvogedu |
| 50 | nabivonabivonabivogedunagevogenaduvogedunanabivobiduvogedu |
| 60 | nabivonagevogedunagevogenanabivobiduvogedu |

90 nabivonaduvogedunagevodunanabivobiduvogedu
100 nabivonabivobinagevogenaduvogedunagevodunanabivobiduvogedu

Care este regula de încifrare? Ce numere reprezintă următoarele coduri cifrate:
nagevonagevogedunanabivobiduvogedu;
nagevonaduvogedunanabivobiduvogedu;
naduvogenanabivobiduvogedu;
nanabivogeduvogedu;
nabivonabivonaduvogedunagevonagevogedunanabivobiduvogedu;
nanagevobiduvogedu?
Încifrați numerele 256 şi 1024 prin acestă metodă.

2. **(!!!) Altfel de codificare binară a numerelor.** Descoperiți metoda de codificare binară a numerelor folosită în continuare:

| | | | |
|---|---|---|---|
| 1 | 1 | 20 | 101010 |
| 2 | 10 | 25 | 1000101 |
| 3 | 11 | 30 | 1010001 |
| 5 | 110 | 40 | 10001001 |
| 10 | 1110 | 50 | 10100100 |
| 15 | 10010 | 60 | 100001000 |

Puteți spune ce numere sînt codificate prin 100, 101, 1000, 1111, 10000 şi 11111 ? Puteți codifica numerele 70, 80, 90, 100, 120, 150 şi 1000 ?

3. **(!!!) Problema dialogului perplex.** Există două numere m şi n din intervalul [2..99] şi două persoane P şi S astfel încît persoana P ştie produsul lor, iar S ştie suma lor. Ştiind că între P şi S a avut loc următorul dialog:
"Nu ştiu numerele" spune P.
"Ştiam ca nu ştii" răspunde S, "nici eu nu ştiu."
"Acuma ştiu !" zice P strălucind de bucurie.
"Acum ştiu şi eu…" şopteşte satisfăcut S.
să se determine toate perechile de numere m şi n ce "satisfac" acest dialog (sînt soluții ale problemei).

4. **(!!!!) Împăturirea celor 8 pătrate.** Împăturiți inițial în opt o foaie dreptunghiulară după care desfaceți-o şi însemnați fiecare pătrățel obținut cu o cifră de la 1 la 8. Proiectați un algoritm şi realizați un program care, primind configurația (numerotarea) celor 8 pătrățele, să poată decide dacă se poate împături foaia astfel obținută reducînd-o de opt ori (la un singur pătrat) astfel încît trecînd cu un ac prin cele opt foi suprapuse acesta să le perforeze exact în ordinea 1, 2, 3, …, 8.

5. **(!!!!) Problema fetelor de la pension.** Problema a apărut pe vremea cînd fetele învățau la pension fără ca prin prezența lor băieții să le tulbure educația. Pedagoaga fetelor unui pension de 15 fete a hotarît ca în fiecare dupa-amiază, la ora de plimbare, fetele să se plimbe în cinci grupuri de cîte trei. Se cere să se stabilească o programare a plimbărilor pe durata unei săptămîni (şapte zile) astfel încît fiecare fată să ajungă să se plimbe numai o singură dată cu oricare din celelalte paisprezece (oricare două fete să nu se plimbe de două ori împreună în decursul unei săptămîni).

# Noțiuni fundamentale de programare

Programarea este disciplina informatică ce are ca scop realizarea de programe care să constituie soluțiile oferite cu ajutorul calculatorului unor probleme concrete. Programatorii sînt acele persoane capabile să implementeze într-un limbaj de programare metoda sau algoritmul propus ca soluție respectivei probleme, ce se pretează a fi soluționată cu ajutorul calculatorului. După nivelul de implicare în efortul de rezolvare a problemelor specialiştii în programare pot fi împărțiți în diverse categorii: analişti, analişti-programatori, ingineri-programatori, simpli programatori, etc. Cu toții au însă în comun faptul că fiecare trebuie să cunoască cît mai bine programare şi să fie capabil, nu doar să citească, ci chiar să scrie "codul sursă", adică programul propriu-zis. Din acest punct de vedere cunoştințele de programare sînt considerate "ABC-ul" informaticii şi sînt indispensabile oricărui profesionist în domeniu.

## 1.Cele trei etape ale rezolvării unei probleme cu ajutorul calculatorului

În rezolvarea sa cu ajutorul calculatorului orice problemă trece prin trei etape obligatorii: *Analiza problemei, Proiectarea algoritmului de soluționare* şi *Implementarea algoritmului într-un program pe calculator.* În ultima etapă, sub acelaşi nume, au fost incluse în plus două subetape cunoscute sub numele de *Testarea* şi *Întreținerea programului.* Aceste subetape nu lipsesc din "ciclul de viață" a oricărui produs-program ce "se respectă" dar , pentru simplificare, în continuare ne vom referi doar la primele trei mari etape.

Dacă etapa *implementării* algoritmului într-un program executabil este o etapă exclusiv practică, realizată "în fața calculatorului", celelalte două etape au un pronunțat caracter teoretic. În consecință, primele două etape sînt caracterizate de un anumit grad de abstractizare. Din punct de vedere practic însă, şi în ultimă instanță, criteriul decisiv ce conferă succesul rezolvării problemei este dat de calitatea implementării propriuzise. Mai exact, succesul soluționării este dat de performanțele programului: utilitate, viteză de execuție, fiabilitate, posibilități de dezvoltare ulterioare, lizibilitate, etc. Cu toate acestea este imatură şi neprofesională "strategia" programatorilor începători care, neglijînd primele două etape, sar direct la a treia fugind de analiză şi de componenta abstractă a efortului de soluționare. Ei se justifică cu toții prin expresii puerile de genul: "Eu nu vreau să mai pierd vremea cu "teoria", am să fac programul cum ştiu eu. Cîtă vreme nu va face altcineva altul mai bun decît al meu, nu am de ce să-mi mai bat capul !".

## 2.Cum se stabileşte corectitudinea şi eficiența soluționării ?

Este adevărat că ultima etapă în rezolvarea unei probleme – implementarea – este decisivă şi doveditoare, dar primele două etape au o importanță capitală. Ele sînt singurele ce pot oferi răspunsuri corecte la următoarele întrebări dificile: *Avem certitudinea că soluția găsită este corectă ? Avem certitudinea că problema este complet rezolvată ? Cît de eficientă este soluția găsită ? Cît de departe este soluția aleasă de o soluție optimă ?*

Să menționăm în plus că literatura informatică de specialitate conține un număr impresionant de probleme "capcană" pentru începători, şi nu numai pentru ei. Ele provin majoritatea din realitatea imediată dar pentru fiecare dintre ele nu se cunosc soluții eficiente. De exemplu, este dovedit teoretic că problema, "aparent banală" pentru un calculator, a proiectării *Orarului optim* într-o instituție de învățămînt (şcoală, liceu, facultate) este o problemă intratabilă la ora actuală (toate programele care s-au realizat pînă acum nu oferă decît soluții aproximative fără a putea spune cît de aproape sau de departe este soluția optimă de orar).

Cîți dintre programatorii începători n-ar fi surprinşi să afle că problema "atît de simplă" (ca enunț), a cărei soluționare tocmai au abandonat-o, este de fapt o problemă dovedită teoretic ca fiind intratabilă sau chiar insolvabilă algoritmic ? Partea proastă a lucrurilor este că, aşa cum ciupercile otrăvite nu pot fi cu uşurință deosebite de cele comestibile, tot astfel problemele netratabile pot fi cu uşurință confundate cu nişte probleme uşoare la o privire rapidă şi lipsită de experiență.

Dacă ar fi să sintetizăm în cîte un cuvînt efortul asupra căruia se concentrează fiecare din cele trei etape – *analiza, proiectarea* şi *implementarea*– cele trei cuvinte ar fi: *corectitudine, eficiență* şi *impecabilitate*. Etapa de analiză este singura care permite *dovedirea cu argumente riguroase* a

corectitudinii soluției, iar etapa de proiectare este singura care poate *oferi argumente precise* în favoarea eficienței soluției propuse.

În general problemele concrete din informatică au în forma lor inițială sau în enunț o caracteristică pragmatică, fiind foarte ancorate în realitatea imediată. Totuşi ele conțin în formularea lor inițială un grad mare de *eterogenitate*, *diversitate* şi *lipsă de rigoare.* Fiecare dintre aceste "defecte" este un obstacol major pentru demonstrarea corectitudinii soluției. Rolul esențial al etapei de analiză este acela de a transfera problema "de pe nisipurile mişcătoare" ale realității imediate de unde ea provine într-un plan abstract, adică de *a o modela*. Acest "univers paralel abstract" este dotat cu mai multă rigoare şi disciplină internă, având legi precise, şi poate oferi instrumentele logice şi formale necesare pentru demonstrarea riguroasă a corectitudinii soluției problemei. Planul abstract în care sînt "transportate" toate problemele de informatică este planul sau *universul obiectelor matematice* iar corespondentul problemei în acest plan va fi *modelul matematic abstract* asociat problemei. Demonstrarea corectitudinii proprietăților ce leagă obiectele universului matematic a fost şi este sarcina matematicienilor. Celui ce analizează problema din punct de vedere informatic îi revine sarcina (nu tocmai uşoară) de a dovedi printr-o **demonstrație constructivă** că există o *corespondență precisă (o bijecție !)* între părțile componente ale problemei reale, "dezasamblată" în timpul analizei, şi părțile componente ale modelului abstract asociat. Odată descoperită, formulată precis şi dovedită, această "perfectă oglindire" a problemei reale în planul obiectelor matematice oferă certitudinea că toate proprietățile şi legăturile ce există între subansamblele modelului abstract se vor regăsii precis (prin reflectare) între părțile interne ale problemei reale, şi invers. Atunci, soluției abstracte descoperite cu ajutorul modelului matematic abstract îi va corespunde o soluție reală concretizată printr-un algoritm ce poate fi implementat într-un program executabil.

Aceasta este calea generală de rezolvare a problemelor şi oricine poate verifica acest fapt. De exemplu, ca şi exercițiu, încercați să demonstrați corectitudinea (adică să se aducă argumente precise, clare şi convingătoare în favoarea *corectitudinii*) metodei de extragere a radicalului învățată încă din şcoala primară (cu grupare cifrelor numărului în grupuri cîte două, etc…) sau a algoritmului lui Euclid de determinare a celui mai mare divizor comun a două numere prin împărțiri întregi repetate. Desigur nu pot fi acceptate argumente copilăreşti de forma: "Algoritmul este corect pentru că aşa l-am învățat!" sau "Este corect pentru că aşa face toată lumea !" din moment ce nu se oferă o argumentație matematică riguroasă.

Ideea centrală a etapei a doua – proiectarea unui algoritm de soluționare eficient poate fi formulată astfel: din studiul *proprietăților* şi *limitelor* modelului matematic abstract asociat problemei se deduc *limitele inferioare ale complexității minimale ("efortului minimal obligatoriu")* inerente **oricărui** algoritm ce va soluționa problema în cauză. *Complexitatea internă a modelului abstract* şi *complexitatea soluției abstracte* se va reflecta imediat asupra *complexității reale a algoritmului*, adică asupra *eficienței* de soluționare a problemei. Acest fapt permite *prognosticarea* încă din această fază – faza de proiectare a algoritmului de soluționare – a eficienței practice, măsurabilă ca *durată de execuție*, a programului.

## 3. Noțiunile fundamentale ale programării: algoritm, limbaje de descriere a algoritmilor, program, limbaje de programare

### 3.1. Algoritmul

Se ştie că la baza oricărui program stă un algoritm (care, uneori, este numit *metodă de rezolvare*). Noțiunea de *algoritm* este o noțiune fundamentală în informatică şi înțelegerea ei, alături de înțelegerea modului de funcționare a unui calculator, permite înțelegerea noțiunii de *program* executabil. Vom oferi în continuare o definiție unanim acceptată pentru noțiunea de *algoritm*:

***Definiție*.** Prin algoritm se înțelege o *mulțime finită* de operații (instrucțiuni) elementare care executate într-o *ordine bine stabilită (determinată)*, pornind de la un set de date de intrare dintr-un domeniu de valori posibile (valide), produce în *timp finit* un set de date de ieşire (rezultate).

***Cele trei caracteristici*** esențiale ale unui algoritm sînt:

1. ***Determinismul*** – dat de faptul că ordinea de execuție a instrucțiunilor algoritmului este bine precizată (strict determinată).
   Acest fapt dă una din calitățile de bază a calculatorului: "el" va face întotdeauna ceea ce i s-a cerut (prin program) să facă, "el" nu va avea inițiative sau opțiuni proprii, "el" nu-şi permite să greşească nici măcar odată, "el" nu se va plictisi ci va duce programul la acelaşi sfîrşit indiferent de cîte ori i se va cere să repete acest lucru. Nu aceeaşi situație se întîmplă cu ființele umane (*Errare humanum est)*. Oamenii pot avea în situații determinate un comportament non-deterministic (surprinzător). Acesta este motivul pentru care numeroşi utilizatori de calculatoare (de exemplu contabilii), datorită fenomenului de personificare a calculatorului (confundarea acțiunilor şi dialogului "simulat" de programul ce rulează pe calculator cu reacțiile unei personalități vii), nu recunosc perfectul determinism ce stă la baza executării oricărui program pe calculator. Exprimîndu-se prin propoziții de felul: "*De trei ori i-am dat să facă calculele şi de fiecare dată mi-a scos aceleaşi valori aiurea!*" ei îşi trădează propria viziune personificatoare asupra unui fenomen determinist.
2. ***Universalitatea*** – dată de faptul că, privind algoritmul ca pe o metodă automată (mecanică) de rezolvare, această metodă are un caracter general-universal. Algoritmul nu oferă o soluție punctuală, pentru un singur set de date de intrare, ci oferă soluție pentru o mulțime foarte largă (de cele mai multe ori infinită) de date de intrare valide. Aceasta este trăsătura de bază care explică deosebita utilitate a calculatoarelor şi datorită acestei trăsături sîntem siguri că investiția financiară făcută prin cumpărarea unui calculator şi a produsului-soft necesar va putea fi cu siguranță amortizată. Cheltuiala se face o singură dată în timp ce programul pe calculator va putea fi executat rapid şi economicos de un număr oricît de mare de ori, pe date diferite !
   De exemplu, metoda (algoritmul) de rezolvare învățată la liceu a ecuațiilor de gradul doi: $ax^2+bx+c=0$, se aplică cu succes pentru o mulțime infinită de date de intrare: $(a,b,c)\in \Re\setminus\{0\}x\Re x\Re$.
3. ***Finitudinea*** – pentru fiecare intrare validă orice algoritm trebuie să conducă în timp finit (după un număr finit de paşi) la un rezultat. Această caracteristică este analogă proprietății de convergență a unor metode din matematică: trebuie să avem garanția, dinainte de a aplica metoda (algoritmul), că metoda se termină cu succes (ea converge către soluție).
   Să observăm şi diferența: în timp ce metoda matematică este corectă chiar dacă ea converge către soluție doar *la infinit* (!), un algoritm trebuie să întoarcă rezultatul după un număr finit de paşi. Să observăm deasemenea că, acolo unde matematica nu oferă dovada, algoritmul nu va fi capabil să o ofere nici el. De exemplu, nu este greu de scris un algoritm care să verifice corectitudinea *Conjecturii lui Goldbach: "Orice număr par se scrie ca sumă de două numere prime"*, dar, deşi programul rezultat poate fi lăsat să ruleze pînă la valori extrem de mari, fără să apară nici un contra-exemplu, totuşi conjectura nu poate fi astfel infirmată (dar nici afirmată!).

### 3.2. Descrierea algoritmilor

Două dintre metodele clasice de descriere a algoritmilor sînt denumite *Schemele logice* şi *Pseudo-Codul.* Ambele metode de descriere conțin doar patru operații (instrucțiuni) elementare care au fiecare un corespondent atît schemă logică cît şi în pseudo-cod.

În cele ce urmează vom înşira doar varianta oferită de pseudo-cod întrucît folosirea schemelor logice s-a redus drastic în ultimii ani. Schemele logice mai pot fi întîlnite sub numele de *diagrame de proces* în anumite cărți de specialitate inginereşti. Avantajul descrierii algoritmilor prin scheme logice este dat de libertatea totală de înlănțuire a operațiilor (practic, săgeata care descrie ordinea de execuție, pleacă de la o operație şi poate fi trasată înspre orice altă operație). Este demonstrat matematic riguros că descrierea prin pseudo-cod, deşi pare mult mai restrictivă (operațiile nu pot fi înlănțuite oricum, ci trebuie executate în ordinea citirii: de sus în jos şi de la stînga la dreapta), este totuşi perfect echivalentă. Deci, este dovedit că plusul de ordine, rigoare şi simplitate pe care îl oferă descrierea prin pseudo-cod nu îngrădeşte prin nimic libertatea programării. Totuşi, programele scrise în limbajele de asamblare, care sînt mult mai compacte şi au dimensiunile mult reduse, nu ar putea fi descrise altfel decît prin scheme logice.

1. ***Atribuirea*** – var:=expresie;

2. ***Intrare/Ieşire*** – *Citeşte* $var_1$, $var_2$, $var_3$, …;
   *Scrie* $var_1$, $var_2$, $var_3$, …; sau *Scrie* $expresia_1$, $expresia_2$, $expresia_3$,…;

3. ***Condiționala -*** *Dacă* <condiție_logică> *atunci* instrucțiune$_1$ [*altfel* instrucțiune$_2$];

4. ***Ciclurile*** – Există (din motive de ușurință a descrierii algoritmilor) trei tipuri de instrucțiuni de ciclare. Ele sînt echivalente între ele, oricare variantă de descriere putînd fi folosită în locul celorlalte două, cu modificări sau adăugiri minimale:

*Repetă* instrucțiune$_1$, instrucțiune$_2$, … *pînă cînd* <condiție_logică>;

*Cît timp* <condiție_logică> *execută* instrucțiune;

*Pentru* var_contor*:*=val_inițială *pînă la* val_finală *execută* instrucțiune;

În cazul ciclurilor, grupul instrucțiunilor ce se repetă se numeşte *corpul ciclului* iar condiția logică care (asemenea semaforului de circulație) permite sau nu reluarea execuției ciclului este denumită *condiția de ciclare* sau *condiția de scurt-circuitare* (după caz). Observăm că ciclul de tipul *Repetă* are condiția de repetare la sfîrşit ceea ce are ca şi consecință faptul că corpul ciclului se execută cel puțin odată, în mod obligatoriu, înainte de verificarea condiției logice. Nu acelaşi lucru se întîmplă în cazul ciclului de tipul *Cît timp,* cînd este posibil ca instrucțiunea compusă din corpul ciclului să nu poată fi executată nici măcar odată. În plus, să mai observăm că ciclul de tipul *Pentru ... pînă la* conține (în mod ascuns) o instrucțiune de incrementare a variabilei contor.

În limba engleză, cea pe care se bazează toate limbajele actuale de programare acestor instrucțiuni, exprimate în limba română, le corespund respectiv: *2. Read, Write; 3. If-Then-Else; 4. Repeat-Until, Do-While, For.* Să observăm că, mai ales pentru un vorbitor de limbă engleză, programele scrise într-un limbaj de programare ce cuprinde aceste instrucțiuni este foarte uşor de citit şi de înțeles, el fiind foarte apropiat de scrierea naturală. Limbajele de programare care sînt relativ apropiate de limbajele naturale sînt denumite *limbaje de nivel înalt (high-level)*, de exemplu limbajul *Pascal*, spre deosebire de limbajele de programare mai apropiate de codurile numerice ale instrucțiunilor microprocesorului. Acestea din urmă se numesc *limbaje de nivel scăzut (low-level)*, de exemplu limbajul de asamblare. Limbajul de programare C are un statut mai special el putînd fi privit, datorită structurii sale, ca făcînd parte din ambele categorii.

Peste tot unde în pseudo-cod apare cuvîntul *instrucțiune* el poate fi înlocuit cu oricare din cele patru instrucțiuni elementare. Această substituire poartă numele de *imbricare* (de la englezescul *brick*-cărămidă). Prin instrucțiune se va înțelege atunci, fie o singură instrucțiune simplă (una din cele patru), fie o *instrucțiune compusă*. Instrucțiunea compusă este formată dintr-un grup de instrucțiuni delimitate şi grupate în mod precis (între acolade *{ }* în C sau între *begin* şi *end* în Pascal).

Spre deosebire de pseudo-cod care permite doar structurile noi formate prin imbricarea repetată a celor patru instrucțiuni (cărămizi) în modul precizat, schemele logice permit structurarea în orice succesiune a celor patru instrucțiuni elementare, ordinea lor de execuție fiind dată de sensul săgeților. Repetăm că deşi, aparent, pseudo-codul limitează libertatea de descriere doar la structurile prezentate, o teoremă fundamentală pentru programare afirmă că puterea de descriere a pseudo-limbajului este aceeaşi cu cea a schemelor logice.

Forma de programare care se bazează doar pe cele patru structuri se numeşte *programare structurată* (spre deosebire de programarea nestructurată bazată pe descrierea prin scheme logice). *Teorema de echivalență a puterii de descriere prin pseudo-cod cu puterea de descriere prin schemă logică* afirmă că programarea structurată (aparent limitată de cele patru structuri) este echivalentă cu programarea nestructurată (liberă de structuri impuse). Evident, prin ordinea, lizibilitatea şi fiabilitatea oferită de cele patru structuri elementare (şi asta fără a îngrădi libertatea de exprimare) programarea structurată este net avantajoasă. În fapt, limbajele de programare nestructurată (*Fortran, Basic)* au fost de mult scoase din uz, ele (limbajele de asamblare) sînt necesare a fi folosite în continuare doar în programarea de sistem şi în programarea industrială (în automatizări).

## 3.3 Programul

Prin *program* se înțelege un şir de instrucțiuni-maşină care sînt rezultatul compilării algoritmului proiectat spre rezolvarea problemei dorite ce a fost descris într-un limbaj de programare (ca şi cod sursă).

Etapele realizării unui program sînt:

- *Editarea codului sursă*, etapă ce se realizează cu ajutorul unui program editor de texte rezultatul fiind un fişier *Pascal* sau *C,* cu extensia *.pas* sau *.c* (*.cpp*)
- *Compilarea*, etapa de traducere din limbajul de programare Pascal sau C în limbajul intern al micro-procesorului, şi este realizată cu ajutorul programului compilator Pascal sau C şi are ca rezultat un fişier obiect, cu extensia *.obj* (în limbajul C) sau *.exe* (în limbajul Pascal)
- *Link-editarea*, etapă la care se adaugă modului obiect rezultat la compilare diferite module conţinînd subprograme şi rutine de bibliotecă, rezultînd un fişier executabil (această etapă este comasată în *Turbo Pascal* sau *Borland Pascal* cu etapa de compilare), cu extensia *.exe*
- *Execuţia (Run)*, etapa de lansare în execuţie propriu-zisă a programului obţinut, lansare realizată de interpretorul de comenzi al sistemului de operare (*command.com* pentru sistemele DOS+Windows)

Observăm că aceste patru (sau trei, pentru Turbo Pascal) etape sînt complet independente în timp unele de altele şi necesită utilizarea a patru programe ajutătoare: Editor de texte, Compilator Pascal sau C, Link-editor şi Interpretorul de comenzi al S.O. În cazul mediilor de programare integrate (Turbo sau Borland) comandarea acestor patru programe ajutătoare precum şi depanarea erorilor de execuţie este mult facilitată.

Deasemenea, merită subliniat faptul că în timp ce fişierul text Pascal sau C, ce conţine codul sursă, poate fi transportat pe orice maşină (calculator) indiferent de micro-procesorul acesteia urmînd a fi compilat "la faţa locului", în cazul fişierului obiect acesta nu mai poate fi folosit decît pe maşina (calculatorul) pentru care a fost creat (datorită instrucţiunilor specifice micro-procesorului din care este compus). Deci, pe calculatoare diferite (avînd micro-procesoare diferite) vom avea nevoie de compilatoare Pascal sau C diferite.

În plus, să remarcăm faptul că fişierele obiect rezultate în urma compilării pot fi link-editate (cu grijă !) împreună chiar dacă provin din limbaje de programare diferite. Astfel, un program rezultat (un fişier *.exe* sau *.com*) poate fi compus din module obiect care provin din surse diferite (fişiere Pascal, C, asamblare, etc.).

## 4. Secretul învăţării rapide a programării

Există posibilitatea învăţării rapide a programării ?

Desigur. Experienţa predării şi învăţării programării ne-a dovedit că există metode diferite de învăţare a programării, mai rapide sau mai lente, mai temeinice sau mai superficiale. Din moment ce se doreşte învăţarea rapidă a programării înseamnă că, pentru cel ce doreşte aceasta, problemele ce îşi aşteaptă rezolvarea cu ajutorul calculatorului sînt importante sau stringente. Am putea chiar presupune că soluţionarea lor rapidă este un deziderat mai important decît învăţarea programării. Tocmai de aceea, fiind conştienţi de acest fapt, vom prezenta în continuare una din cele mai rapide metode de învăţare a programării.

Să observăm mai întîi că pentru învăţarea unei limbi străine este necesară comunicarea şi vorbirea intensă a acelei limbi. Cu toţii am putut constata că dacă există o motivaţie sau nevoie puternică de a comunica în acea limbă, cel puţin pentru o perioadă de timp, procesul de învăţare a ei este foarte rapid. De exemplu, dacă ne aflăm într-o ţară străină sau dacă dorim apropierea de o persoană străină (mai ales dacă este atrăgătoare şi de sex opus…) categoric vom constata că am învăţat mult mai iute limba respectivă. Şi aceasta datorită faptului că efortul de învăţare a fost mascat în spatele efortului (intens motivat!) de a comunica şi de a ne face cunoscute intenţiile şi gîndurile.

La fel, pentru învăţarea rapidă şi cu uşurinţă a programării efortul trebuie îndreptat, nu spre "silabisirea" limbajului de programare, ci spre rezolvarea de probleme şi spre scrierea directă a programelor de soluţionare a acestora. Concentrîndu-ne asupra problemelor ce le soluţionăm nici nu vom observa cînd şi în ce fel am învăţat să scriem programe. La urma urmei, programarea este doar un instrument, doar o unealtă "de scris", şi nu un scop în sine. Dacă vrei iute să înveţi să scrii, contează cum sau în ce mînă ţii stiloul ?…

Nu trebuie deloc neglijat şi un al doilea "factor secret". Aşa cum "*meseria nu se învaţă, ci se fură*", tot astfel programarea se poate învăţa mult mai uşor apelînd la ajutorul unui profesor sau a unui specialist. Acesta, prin experienţa şi cunoştinţele sale de specialitate ne poate ajuta să păşim alături de el "pe cărări bătătorite" şi într-un ritm susţinut.

În concluzie, într-o descriere plastică şi metaforică, metoda secretă cea mai rapidă de "ascensiune" în programare este metoda *"privirii concentrate spre vîrf, cu ghidul alături şi pe cărări bătătorite"*.

# Noțiuni primare de programare în Pascal şi C

În spiritul celor spuse mai sus, vom introduce acum "într-un ritm alert", prin exemple concrete, noțiunile elementare de programare în limbajele Pascal şi C (în paralel). Vom pleca de la prezentarea structurii generale a unui program iar apoi vom trece la prezentarea celor patru structuri-instrucțiuni elementare conținute în psedo-limbajul de descriere a algoritmilor. Vom avea în plus grijă de a precede descrierea fiecărei structuri elementare de liniile de declarare a tipului variabilelor implicate. Peste tot vor apare linii de comentariu (ignorate de compilator). În limbajul Pascal comentariile sînt cuprinse între acolade {*comentariu*}, pe cînd în C ele sînt cuprinde între construcția de tipul /* *comentariu**/ sau apar la sfîrşitul liniei precedate de două slash-uri //*comentariu*.

<table>
<tr><td colspan="2">Structura unui program</td></tr>
<tr><td>Program Nume_de_Program; {această linie poate să lipsească}<br>{ Zona de declarații constante, variabile, proceduri şi funcții }<br>BEGIN<br>{ Corpul programului format din instrucțiuni terminate cu punct-vigulă ; Corpul programului poate fi privit ca o instrucțiune compusă }<br>END.<br>(Orice se va scrie după punct va fi ignorat de către compilator)</td><td>// linii de incluziuni de fişiere header<br><br>// declarații de variabile şi funcții externe (globale)<br><br>void main(void){<br>// declarații de variabile locale<br><br>// corpul programului format din instrucțiuni terminate cu punct-vigulă ;<br>}</td></tr>
<tr><td>Exemplu :<br>Program Un_Simplu_Test;<br>Const e=2.68;<br>Var x:real;<br>BEGIN<br>  x:=1./2+e*(1+e);<br>  Writeln(‘Rezultatul este:’,x);<br>END.</td><td>Exemplu :<br>#include <stdio.h><br><br>int e=2.68;<br>float x;<br>void main(void){<br>  x=1./2+e*(1+e);<br>  printf(“Rezultatul este %f:”,x);<br>}</td></tr>
<tr><td colspan="2">Atribuirea : var:=expresie;</td></tr>
<tr><td>Var i,j:integer;perimetrul:real;<br>……..<br>  j:=2000 div 15;  { împărțire întreagă obligatorie }<br>  i:=i+(j-1)*Sqr(2*j+1); { Sqr (Square) – funcția de ridicare la pătrat }<br>  perimetrul:=2*PI*i; { PI – constantă reală implicită }</td><td>#include <math.h>  // declară constanta M_PI<br>int i,j; float perimetrul;<br>……..<br>  j=2000 / 15;  // împărțire întreagă implicită !!<br>  i+=(j-1)* (2*j+1)*(2*j+1); // în C avem operatorul<br>// de adunare + înainte de egal = ; funcția putere în<br>// C este pow(x,y)<br>  perimetrul=2*M_PI*i;</td></tr>
<tr><td colspan="2">Intrare/Ieşire :<br>Citeşte var<sub>1</sub>, var<sub>2</sub>, var<sub>3</sub>, …;<br>Scrie var<sub>1</sub>, var<sub>2</sub>, var<sub>3</sub>, …;<br>Sau<br>Scrie expresia<sub>1</sub>, expresia<sub>2</sub>, expresia<sub>3</sub>,…;</td></tr>
<tr><td>Var i,j:integer;perimetrul:real;<br>……..<br>  Readln(i,j); { citirea variabilelor i şi j }<br>  Perimetrul:=2*PI*i;<br>  Writeln(‘Raza=’,i:4,’ Perimetrul=’,perimetrul:6:2,’ Aria=’, PI*Sqr(i):6:2);<br>{ perimetrul si aria fiind valori reale, se afiseaza cu descriptorul de format de afisare :6:2 – pe 6 poziții de ecran cu rotunjit la 2 zecimale }</td><td>#include <math.h>  // declară constanta M_PI<br>int i,j; float perimetrul;<br>……..<br>scanf(“%i %i”,&i,&j); // “%i %i” este descriptorul de format de citire, & este operatorul de adresare<br>perimetrul=2*M_PI *i;<br>printf(“Raza=%4i Perimetrul= %6.2f Aria= %6.2f”,i,perimetrul,M_PI*i*i); // %6.2f – descriptorul de format de afisare a unei valori reale(flotante) pe 6 poziții rotunjit la 2 zecimale</td></tr>
</table>

| **Condiționala :**<br>**Dacă <condiție_logică> atunci instrucțiune$_1$ [altfel instrucțiune$_2$];** | |
|---|---|
| Var i,j,suma:integer;<br>……..<br>If i <= 2*j+1 then suma:=suma+i<br>else suma:=suma+j; | int i,j,suma;<br>……..<br>if (i<=2*j+1) suma+=i<br>else suma+=j; |
| **Ciclul de tipul Repeat-Until:**<br>**Repetă instrucțiune$_1$, instrucțiune$_2$, … pînă cînd <condiție_logică>;** | |
| Var i,j,suma:integer;<br>……..<br>suma:=0;i:=1;<br>Repeat<br>  suma:=suma+i; i:=i+1;<br>Until i>100; | int i,j,suma;<br>……..<br>suma=0;i=1;<br>do<br>  suma+=i;<br>while (i++<100); |
| **Ciclul de tipul Do-While:**<br>**Cît timp <condiție_logică> execută instrucțiune;** | |
| Var i,j,suma:integer;<br>……..<br>suma:=0;i:=1;<br>While i<=100 do begin<br>  suma:=suma+i; i:=i+1;<br>End; | int i,j,suma;<br>……..<br>suma=0;i=1;<br>while (i++<100)<br>  suma+=i; |
| **Ciclul de tipul For (cu contor):**<br>**Pentru var_contor:=val_inițială pînă la val_finală execută instrucțiune;** | |
| Var i,j,suma:integer;<br>……..<br>suma:=0;<br>For i:=1 to 100 do<br>  Suma:=suma+i; | int i,j, suma;<br>……..<br>for(suma=0,i=1;i<=100;i++)<br>  suma+=i; |

## Exemple de probleme rezolvate

Prezentăm în continuare, spre inițiere, cîteva exemple de probleme rezolvate. Vom oferi programul rezultat atît în limbajul de programare Pascal cît şi în limbajul C. Deasemenea, fiecare program va fi precedat de o scurtă descriere a modului de elaborare a soluției.

**1. Se citesc *a,b,c* coeficienții reali a unei ecuații de gradul II. Să se afişeze soluțile ecuației.**

**Descrierea algoritmului:**
- ecuația de gradul II este de forma $ax^2+bx+c=0$
-presupunînd că $a \neq 0$ calculăm determinantul ecuatiei delta=b*b-4*a*c
- dacă delta >= 0 atunci ecuația are soluțiile reale $x_{1,2}=(-b\pm\sqrt{delta})/(2*a)$
- dacă delta < 0 atunci ecuația are soluțiile complexe $z_1=(-b/(2*a), \sqrt{(-delta)}/(2*a))$, $z_1=(-b/(2*a), -\sqrt{(-delta)}/(2*a))$

```
Program Ecuatie_grad_2;  { varianta Pascal }
Var a,b,c,delta:real;

BEGIN
Write('Introd. a,b,c:');Readln(a,b,c);
delta:=b*b-4*a*c;
If  delta>=0 then
      Begin
        Writeln('x1=',(-b-sqrt(delta))/(2*a):6:2);
        Writeln('x2=',(-b+sqrt(delta))/(2*a):6:2);
```

```
      End
   else Begin
        Writeln('z1=(',-b/(2*a):6:2, ‘,’ , -sqrt(-delta))/(2*a):6:2, ‘)’);
        Writeln('z2=(', -b/(2*a):6:2, ‘,’ , sqrt(-delta))/(2*a):6:2, ‘)’);
      End
Readln;
END.
```

**// versiunea C**

```
#include <stdio.h>
#include <math.h>

float a,b,c;  // coeficientii ecuatiei de gradul II
float delta;

void main(){
printf("Introd.coefic.ecuatiei a b c:");scanf("%f %f %f",&a,&b,&c);
delta=b*b-4*a*c;
if (delta>=0) {
   printf("Sol.reale: x1=%6.2f, x2=%6.2f",(-b+sqrt(delta))/2./a,(-b-sqrt(delta))/2./a);
} else {
  printf("Sol.complexe: x1=(%6.2f,%6.2f), x2=(%6.2f,%6.2f)",-b/2./a,sqrt(-delta)/2./a,-b/2/a,-sqrt(-
delta)/2./a);
}
}
```

**2. Să se determine dacă trei numere *a,b,c* reale pot reprezenta laturile unui triunghi. Dacă da, să se caculeze perimetrul şi aria sa.**

**Descrierea algoritmului:**
- condiția necesară pentru ca trei numere să poată fi lungimile laturilor unui triunghi este ca cele trei numere să fie pozitive (condiție implicită) şi suma a oricăror două dintre ele să fie mai mare decît cel de-al treilea număr
- după condiția este îndeplinită vom calcula perimetrul şi aria triunghiului folosind **formula lui Heron** s=sqrt(p(p-a)(p-b)(p-c)) unde p=(a+b+c)/2.

```
Program Laturile_Unui_Triunghi;  { Pascal }
Var a,b,c,s,p:real;

function laturi_ok:boolean;
begin
   laturi_ok:= (a>0) and (b>0) and (c>0) and  (a+b>c) and (a+c>b) and (b+c>a) ;
end;

BEGIN
write('introduceti laturile');readln(a,b,c);
IF laturi_ok then
begin
    p:=(a+b+c)/2;
    s:=sqrt(p*(p-a)*(p-b)*(p-c));
    writeln('Aria=',s:5:2);
    writeln(‘Perimetrul=’,2*p:5:2);
end
else writeln('Nu formeaza triunghi');
readln;
END.
```

**// versiunea C**

```
#include <stdio.h>
#include <math.h>

float a,b,c,s,p;

int validare_laturi(float a,float b,float c){
  return( (a>0)&&(b>0)&&(c>0)&&(a+b>c)&&(b+c>a)&&(a+c>b));
}

void main(void){
  printf("Introd.laturile a b c:");scanf("%f  %f  %f",&a,&b,&c);
  if (validare_laturi(a,b,c)){
     p=(a+b+c)/2;s=sqrt(p*(p-a)*(p-b)*(p-c));
     printf("Aria=%6.2f, Perimetrul=%6.2f",s,2*p);
  }
}
```

**3. Se citeşte *n* întreg. Să se determine suma primelor *n* numere naturale.**

**Descrierea algoritmului:**
- vom oferi varianta în care suma primelor n numere naturale va fi calculata cu una dintre instructiunile repetitive cunoscute(for,while ,repeat) fără a apela la formula matematică cunoscută S(n)=n*(n+1)/2

```
Program Suma_n; { Pascal }
Var  n,s,i:word;

BEGIN
Writeln('Introduceti limita n=');Readln(n);
s:=0;
For i:=1 to n do  s:=s+i;
Writeln('s=',s);
Readln;
END.
```

**// versiunea C**

```
#include <stdio.h>

int n,s;

void main(void){
  printf("Introd. n:"); scanf("%i",&n);
 for(;n>0;n--)s+=n;
 printf("S(n)=%i",s);
}
```

**4. Se citeşte valoarea întreagă *p*. Să se determine daca *p* este număr prim.**

**Descrierea algoritmului:**
- un număr p este prim dacă nu are nici un divizor înafară de 1 şi p cu ajutorul unei variabile contor *d* vom parcurge toate valorile intervalului [2.. $\sqrt{p}$]; acest interval este suficient pentru depistarea unui divizor, căci: $d_1 \mid p \Rightarrow p = d_1*d_2$ (unde $d_1 < d_2$) $\Rightarrow d_1 \le \sqrt{d_1*d_2} = \sqrt{p}$ iar $d_2 \ge \sqrt{d_1*d_2} = \sqrt{p}$

```
Program Nr_prim;  { Pascal }
Var p,i:word;
   prim:boolean;

BEGIN
write('p=');readln(p);
prim:=true;
```

```
for i:=2 to trunc(sqrt(p)) do
     if n mod i=0 then  prim:=false;
prim:=true;
if prim then
   write(p,'  este nr prim')
  else
   write(p,'  nu e nr prim');
END.
```

**// versiunea C (optimizată !)**

```
#include <stdio.h>
#include <math.h>

int p,i,prim;

void main(void){
  printf("Introd. p:"); scanf("%i",&p);
  for(i=3, prim=p % 2; (i<=sqrt(p))&&(prim); i+=2)
     prim=p % i;
 printf("%i %s nr.prim", p, (prim ? "este": "nu este"));
}
```

**5. Se citeşte o propoziţie (şir de caractere) terminată cu punct. Să se determine cîte vocale şi cîte consoane conţine propoziţia.**

```
Program Vocale;
Var sir:string[80];
   Vocale,Consoane,i:integer;

BEGIN
  Write('Introd.propozitia terminata cu punct:');Realn(sir);
   i:=1;Vocale:=0;Consoane:=0;
   While sir[i]<>'.' do begin
      If Upcase(sir[i]) in ['A','E','I','O','U'] then Inc(Vocale)
      else If Upcase(sir[i]) in ['A'..'Z'] then Inc(Consoane);
      Inc(i);
   end;
   Writeln('Vocale:',Vocale,' Consoane:', Consoane,' Alte caractere:',i-Vocale-Consoane);
END.
```

**// versiunea C**

```
#include <stdio.h>
#include <ctype.h>

int i,vocale=0,consoane=0;
char c,sir[80];

void main(void){
  printf("Introd.propozitia terminata cu punct:");gets(sir);
  for(i=0;sir[i]!='.';i++)
    switch (toupper(sir[i])){
      case 'A':
               case 'E':
               case 'I':
               case 'O':
               case 'U': vocale++; break;
      default: if (isalpha(sir[i])) consoane++;
    }
```

```
  printf("Vocale:%i, Consoane:%i, Alte car.:%i", vocale, consoane, i-vocale-consoane);
}
```

## Metoda practică de învățare ce garantează rezultate imediate

Dacă cele spuse mai sus cu privire la secretul învățării rapide a programării, acum nu ne mai rămîne decît să începem să aplicăm practic ideile prezentate. Pentru aceasta, avem la dispoziție următoarea metodă care garantează cu siguranță rezultate. Iat-o, pe paşi:

1. se citeşte şi se înțelege cît mai bine exemplul de problemă rezolvată (se poate începe chiar cu primul exemplu de mai sus)
2. se acoperă (se ascunde) soluția şi se încearcă reproducerea ei din memorie (reinventarea soluției) pe calculator
3. numai în cazuri excepționale se poate apela (se poate trage cu ochiul) la soluție

Oricare dintre noi poate recunoaşte aici metoda pe care o aplică copiii din primele clase primare: metoda trasului cu ochiul la rezultatul aflat la spatele manualului sau al culegerii de probleme. Din moment ce metoda este verificată şi garantată (am folosit-o şi noi cîndva), de ce ne-ar fi ruşine s-o aplicăm acum din nou ?

Iată în continuare o listă de probleme de "antrenament" care au majoritea rezolvarea într-unul din capitolele următoare. Este numai bine pentru a începe să aplicăm metoda oferită chiar acum !

## Probleme selecționate - Enunțuri

## Probleme propuse spre rezolvare (probleme de antrenament)

1. Se citesc a, b, c trei variabile reale.
- Să se afişeze maximul şi minimul celor trei numere.
- Să se afişeze cele trei numere în ordine crescătoare.
- Să se determine dacă cele trei numere pot reprezenta laturile unui triunghi. Dacă da, să se determine dacă triunghiul respectiv este isoscel, echilateral sau oarecare.
- Să se determine dacă cele trei numere pot reprezenta laturile unui triunghi. Dacă da, să se determine mărimile unghiurilor sale şi dacă este ascuțit-unghic sau obtuz-unghic.
- Să se afişeze media aritmetică, geometrică şi hiperbolică a celor trei valori.

2. Se citeşte n o valoare întreagă pozitivă.
- Să se determine dacă n este divizibil cu 3 dar nu este divizibil cu 11.
- Să se determine dacă n este pătrat sau cub perfect.
- Să se afişeze primele n pătrate perfecte.
- Să se determine numărul cuburilor perfecte mai mici decît n.
- Să se găsească primul număr prim mai mare decît n.
- Să se afişeze primele n numere prime: 2, 3, 5, 7,…, $p_n$.
- Să se determine toate numerele de 4 cifre divizibile cu n.
- Să se determine suma cifrelor lui n.
- Să se afişeze răsturnatul lui n. (Ex: n=1993 => n_răsturnat =3991).
- Să se afişeze următorul triunghi de numere:

  1
  1 2
  1 2 3
  ……..
  1 2 3 … n

3. Se citesc m, n două variabile întregi pozitive.
   - Să se determine toate pătratele perfecte cuprinse între m şi n, inclusiv.
   - Să se determine toate numerele prime cuprinse între m şi n.
   - Să se determine toate numerele de 4 cifre care se divid atît cu n cît şi cu m.

- Să se determine c.m.m.d.c. al celor două numere folosind algoritmul lui Euclid.

4. Să se calculeze $u_{20}$ , $u_{30}$ , $u_{50}$ ai şirului cu formula recursivă $u_n=1/12u_{n-1}+1/2u_{n-2}$ pentru n>=2 şi $u_0=1$, $u_1=1/2$.

5. Se citeşte n gradul unui polinom şi şirul $a_n$, $a_{n-1}$, … , $a_1$, $a_0$ coeficienților unui polinom P.
   - Se citeşte x, să se determine P(x).
   - Se citesc x şi y, să se determine dacă polinomul P schimbă de semn de la x la y.
   - Se citeşte a, să se determine restul împărțirii lui P la x-a.

6. Se citesc m, n gradele a două polinoame P şi Q, şi coeficienții acestora. Să se determine polinomul produs R=PxQ.

7. Se citeşte o propoziție (şir de caractere) terminată cu punct.
- Să se determine cîte vocale şi cîte consoane conține propoziția.
- Să se afişeze propoziția în ordine inversă şi cu literele inversate (mari cu mici).
- Să se afişeze fiecare cuvînt din propoziție pe cîte o linie separată.
- Să se afişeze propoziția rezultată prin inserarea în spatele fiecărei vocale ‘v’ a şirului “pv” (“vorbirea găinească”).

8. Se citeşte m, n dimensiunea unei matrici $A=(a_{i,j})_{mxn}$ de valori reale.
- Se citesc l, c. Să se afişeze matricea obținută prin eliminarea liniei l şi a coloanei c.
- Se citeşte n întreg pozitiv, să se afişeze matricea obținută prin permutarea circulară a liniilor matricii cu n poziții.
- Să se determine suma elementelor pe fiecare linie şi coloană.
- Să se determine numărul elementelor pozitive şi negative din matrice.
- Să se determine linia şi coloana în care se află valoarea maximă din matrice.
- Să se determine linia care are suma elementelor maximă.

9. Se citesc m, n, p şi apoi se citesc două matrici $A=(a_{i,j})_{mxn}$ şi $B=(b_{j,k})_{nxp}$.Să se determine matricea produs C=AxB.

10. Se citeşte un fişier ce conține mai multe linii de text.
- Să se afişeze linia care are lungime minimă.
- Să se afişeze liniile care conțin un anumit cuvînt citit în prealabil.
- Să se creeze un fişier care are acelaşi conținut dar în ordine inversă.

## Probleme de examen

1. Se citeşte x o valoarea reală. Să se determine radical(x) cu 5 zecimale exacte pe baza şirului convergent $x_n=1/2\ (x_{n-1}+x / x_{n-1})$ cu $x_0>0$ arbitrar ales.
2. Se citeşte x o valoarea reală şi k un număr natural. Să se determine radical de ordinul k din x cu 5 zecimale exacte pe baza şirului convergent $x_n=1/k\ (\ (k-1)\ x_{n-1}+x / x_{n-1}^{k-1})$ cu $x_0>0$ arbitrar ales.
3. Să se determine c.m.m.m.c. a două numere m, n citite.
4. Se citeşte n, să se determine toate perechile (x, y) care au cmmmc(x,y)=n.
5. Se citesc a, b, c întregi pozitive, să se determine toate perechile întregi (x, y) care conduc la egalitatea c=ax+by.
6. Se citeşte n o valoare întreagă pozitivă. Să se determine toate descompunerile în diferență de pătrate a lui n.
7. Să se determine toate tripletele (i, j, k) de numere naturale ce verifică relația $i^2+j^2+k^2=n$ unde n se citeşte.
8. Se citeşte n, să se afişeze toate numerele pitagoreice mai mici sau egale cu n.
9. Se citeşte n, să se determine toate numerele perfecte mai mici decît n. (Un număr este perfect dacă este egal cu suma divizorilor săi, ex. 6=1+2+3.)
10. Se citeşte n, să se afişeze toate numerele de n cifre, formate numai cu cifrele 1 şi 2 şi care se divid cu $2^n$.
11. Se citeşte n, să se afişeze toate numerele de n cifre care adunate cu răsturnatul lor dau un pătrat perfect.
12. Se citeşte n întreg pozitiv, să se afişeze n transcris în baza 2.
13. Se citeşte n întreg pozitiv scris în baza 2, să se afişeze n transcris în baza 10.
14. Se citeşte n întreg pozitiv, să se afişeze n în transcripția romană. (Ex: 1993=MCMXCIII , unde M=1000, D=500, C=100, L=50, X=10, V=5, I=1.)
15. Se citeşte n, să se afişeze descompunerea acestuia în factori primi.
16. Se citesc m, n numărătorul şi numitorul unei fracții. Să se simplifice această fracție.
17. Se citeşte n, să se afişeze toate posibilitățile de scriere a lui n ca sumă de numere consecutive.
18. Se citeşte n şi k, să se afişeze n ca sumă de k numere distincte.
19. Se citeşte n, să se determine o alegere a semnelor + şi – astfel încît să avem relația $1\pm2\pm\ldots\pm(n+1)\pm n=0$, dacă ea este posibilă.
20. Se citeşte n şi şirul de valori reale $x_1, x_2, \ldots, x_{n-1}, x_n$ ordonat crescător. Să se determine distanța maximă între două elemente consecutive din şir.
21. Se citeşte n gradul unui polinom şi şirul $x_n, x_{n-1}, \ldots, x_1$ soluțiilor reale a unui polinom P. Să se determine şirul $a_n, a_{n-1}, \ldots, a_1, a_0$ coeficienților polinomului P.
22. Se citesc două şiruri de valori reale $x_1, x_2, \ldots, x_{n-1}, x_n$ şi $y_1, y_2, \ldots, y_{m-1}, y_m$ ordonate crescător. Să se afişeze şirul $z_1, z_2, \ldots, z_{n+m-1}, z_{n+m}$ rezultat prin interclasarea celor două şiruri.
23. Un şir de fracții ireductibile din intervalul [0,1] cu numitorul mai mic sau egal cu n se numeşte şir Farey de ordinul n. De exemplu, şirul Farey de ordinul 5 (ordonat crescător) este: 0/1, 1/5, ¼, 1/3, 2/5, ½, 3/5, 2/3, ¾, 4/5, 1/1. Să se determine şirul Farey de ordinul n, cu n citit.
24. Se citeşte n şi S o permutare a mulțimii {1, 2, …, n}. Să se determine numărul de inversiuni şi signatura permutării S.
25. Se citeşte n şi S o permutare a mulțimii {1, 2, …, n}. Să se determine cel mai mic număr k pentru care $S^k=\{1, 2, \ldots, n\}$.
26. Fie M={1, 3, 4, …} mulțimea numerelor obținute pe baza regulii R1, şi a regulii R2 aplicate de un număr finit de ori: R1) $1\in M$ R2) Dacă $x\in M$ atunci $y=2x+1$ şi $z=3x+1$ aparțin lui M. Se citeşte n, să se determine dacă n aparține mulțimii M fără a genera toate elementele acesteia mai mici decît n.
27. Se citeşte n, k şi o matrice $A=(a_{i,j})_{nxn}$ pătratică. Să se determine $A^k$.
28. Se citeşte n şi o matrice $A=(a_{i,j})_{nxn}$ pătratică. Să se determine *d* determinantul matricii A.
29. Se citeşte n şi cele n perechi $(x_i, y_i)$ de coordonate a n puncte $P_i$ în plan. Să se determine care dintre cele n puncte poate fi centrul unui cerc acoperitor de rază minimă.
30. Să se determine, cu 5 zecimale exacte, rădăcina ecuației $x^3+x+1=0$ care există şi este unică în intervalul [-1,1].
31. Se citeşte n şi şirul de valori reale $x_1, x_2, \ldots, x_{n-1}, x_n$. Să se determine poziția de început şi lungimea celui mai mare subşir de numere pozitive.
32. Se citeşte n, să se afişeze binomul lui Newton: $(x+y)^n$.

33. Se citeşte n, să se afişeze binomul lui Newton generalizat: $(x_1+x_2+\ldots+x_p)^n=\Sigma n!/(n_1!n_2!\ldots n_p!)$ $x_1^{n_1}x_2^{n_2}\ldots x_p^{n_p}$ pentru $n_1+n_2+\ldots+n_p=n$ şi $n_i>0$, i=1,p.
34. Se citeşte n, să se determine descompunerea lui n ca sumă de numere Fibonacci distincte. ($F_n=F_{n-1}+F_{n-2}$ pentru n>1 şi $F_1=1$, $F_0=0$).
35. Avem la dispoziție următoarele trei operații care se pot efectua asupra unui număr n: O1) i se adaugă la sfîrşit cifra 4; O2) i se adaugă la sfîrşit cifra 0; O3) dacă n este par se împarte la 2. Să se afişeze şirul operațiilor care se aplică succesiv, pornind de la 4, pentru a obține un n care se citeşte.
36. Fie funcția lui Ackermann definită astfel: A(i,n)=n+1 pentru i=0; A(i,n)=A(i-1,1) pentru i>0 şi n=0; A(i,n)=A(i-1,A(i,n-1)) pentru i>0 şi n>0. Care este cea mai mare valoare k pentru care se poate calcula A(k,k) ?
37. Să se determine suma tuturor numerelor formate numai din cifre impare distincte.
38. Scrieți o funcție recursivă pentru a determina c.m.m.d.c. a două numere m şi n.
39. Scrieți o funcție recursivă pentru a calcula $a^n$ pe baza relației $a^n=(a^k)^2$ pentru n=2k, şi $a^n=a(a^k)^2$ pentru n=2k+1.
40. Scrieți o funcție recursivă pentru a determina prezența unui număr x într-un şir de valori reale $x_1$, $x_2$, … , $x_{n-1}$, $x_n$ ordonate crescător folosind algoritmul căutării binare.
41. Scrieți o funcție recursivă pentru a determina o aşezare a 8 turnuri pe o tablă de şah astfel încît să nu se atace între ele. (Tabla de şah va fi reprezentată printr-o matrice pătratică de 8x8).
42. Să se determine peste cîți ani data de azi va cădea în aceeaşi zi a săptămînii.
43. Avem la dispoziție un fişier ce conține numele, prenumele şi media tuturor studenților din grupă.
    - Să se afişeze studentul cu cea mai mare medie.
    - Să se afişeze toți studenții bursieri.
    - Să se afişeze studentul care are media cea mai apropiată de media aritmetică a mediilor pe grupă.
    - Să se afişeze toți studenții din prima jumătate a alfabetului.
    - Să se afişeze toți studenții în ordine inversă decît cea din fişier.
    - Să se creeze un fişier catalog care să conțină aceleaşi informații în ordinea alfabetică a numelui.
44. Avem la dispoziție două fişiere ce conțin numele, prenumele şi media tuturor studenților din cele două grupe ale anului în ordinea descrescătoare a mediilor.
    - Să se afişeze toți studenții din ambele grupe care au media mai mare decît media anului.
    - Să se creeze prin interclasare un fişier totalizator care conține toți studenții anului în ordinea descrescătoare a mediilor.

## Probleme dificile

După cum se poate bănui, informatica conține și ea, la fel ca matematica, o mulțime de probleme foarte dificile care își așteaptă încă rezolvarea. Asemănarea cu matematica ne interesează mai ales în privința unui aspect "capcană" asupra căruia dorim să atragem atenția aici.

Enunțurile problemelor dificile sau foarte dificile de informatică este, în 99% din cazuri, foarte simplu și poate fi citit și înțeles de orice student. Acest fapt consituie o capcană sigură pentru cei ignoranți. Dacă în matematică lucrurile nu stau așa, asta se datorează numai faptului că studiul matematicii are vechime și problemele, împreună cu dificultățile lor, sînt ceva mai bine cunoscute. În informatică nu avem însă aceeași situație. Ba chiar se întîmplă că probleme foarte dificile sînt amestecate în culegerile de probleme de informatică printre probleme ușoare, mai ales datorită lipsei de cultură de specialitate a autorilor.

Acest capitol își propune să pună în gardă în privința dificultății problemelor oferind o mică inițiere în acest domeniu (mai multe se pot afla studiind *Complexitatea algoritmilor și dificultatea problemelor*). Deasemeni el își propune să umple lacuna ce mai există încă la ora actuală în cultura de specialitate.

Dificultatea problemelor de programare a căror enunțuri urmează este considerată maximă de teoreticienii informaticii (ele se numesc *probleme NP-complete*). Nu vă lăsați păcăliți de faptul că le-ați întîlnit în unele culegeri de programare. Ele sînt depășite în dificultate doar de problemele insolvabile algoritmic ! Dar în ce constă dificultatea lor ?

Spre deosebire de matematică, dificultatea problemelor de informatică nu este dată de faptul că nu se cunoaște un algoritm de rezolvare a lor, ci datorită faptului că nu se cunoaște un algoritm *eficient* **(!)** de rezolvare a lor. Existența unei metode de proiectare a algoritmilor atît de general valabilă, cum este metoda back-tracking, face ca prea puține probleme cu care ne putem întîlni să nu aibă o soluție. Dar, întrucît în cazul metodei back-tracking, căutarea soluției se face într-un mod exhaustiv (se caută "peste tot", pentru ca să fim siguri că nu lăsăm nici o posibilitate neexplorată), durata căutării are o creștere exponențial-proporțională cu dimesiunea datelor de intrare. De exemplu, timpul de căutare care depinde de valoarea de intrare $n$ poate avea o expresie de forma $T(n)=c \cdot 2^n$ *secunde*, unde $c$ este un factor de proporționalitate ce poate varia, să zicem, de la $c=12.5$ cînd algoritmul este executat pe un calculator sau $c=62.8$ cînd el este rulat pe un calculator de cinci ori mai performant. Dar, indiferent de calculator, pentru $n=100$ avem $2^{100}=(2^{10})^{10} \approx (10^3)^{10}=10^{30}$, deci timpul măsurat în secunde are ordinul de mărime mai mare de 30. Cea mai largă estimare pentru vîrsta Universului nostru nu depășește *20 mild. ani* ceea ce transformat în secunde conduce la un ordin de mărime mai mic de 20. Deci, chiar și pentru valori mici ale lui $n$ (de ordinul sutelor) am avea de așteptat pentru găsirea soluției de 10 miliarde de ori mai mult decît a trecut de la Big Bang încoace ! Pot fi în această situație considerate astfel de programe ca rezolvări rezonabile, doar pentru că ele găsesc soluția în cazurile în care $n=2, 3, 4, ..., 10$ ?

Exemplele următoare sînt doar cîteva, ușor de întîlnit "din greșeală", dintr-o listă cunoscută ce conține la ora actuală peste șase sute de astfel de probleme. Pentru fiecare din aceste probleme nu li se cunosc alte soluții decît inutilii algoritmi de gen back-tracking. În listă apare des noțiunea de *graf*, așa că o vom introduce în continuare cît mai simplu cu putință: printr-un graf se înțelege o mulțime de *vîrfuri* și o mulțime de *muchii* care unesc unele vîrfuri între ele. Orice hartă (schematizată) rutieră, feroviară sau de trafic aerian reprezintă desenul unui graf.

1. **Problema partiționării sumei.** Fie $C$ un întreg pozitiv și $d_1, d_2, ..., d_n$ o mulțime de $n$ valori întregi pozitive. Se cere să se găsească o partiționare a mulțimii $d_1, d_2, ..., d_n$ astfel încît suma elementelor partiției să fie exact $C$.
2. **Problema rucsacului.** Avem un rucsac de capacitate întreagă pozitivă $C$ și $n$ obiecte cu dimensiunile $d_1, d_2, ..., d_n$ și avînd asociate profiturile $p_1, p_2, ..., p_n$ (în caz că ajung în rucsac). Se cere să se determine profitul maxim ce se poate obține prin încărcarea rucsacului (fără ai depăși capacitatea).
3. **Problema colorării grafului.** Să se determine numărul minim de culori (*numărul cromatic*) necesar pentru colorarea unui graf astfel încît oricare două vîrfuri unite printr-o muchie (*adiacente*) să aibă culori diferite.
4. **Problema împachetării.** Presupunînd că dispunem de un număr suficient de mare de cutii fiecare avînd capacitatea *1* și $n$ obiecte cu dimensiunile $d_1, d_2, ..., d_n$, cu $0<d_i<1$, se cere să se determine numărul optim (cel mai mic) de cutii necesar pentru împachetarea tutror celor $n$ obiecte.
5. **Problema comisului voiajor.** (varianta simplificată) Dîndu-se un graf (o hartă), se cere să se găsească un circuit (un șir de muchii înlănțuite) care trece prin fiecare vîrf o singură dată.

Majoritatea acestor probleme apar ca probleme centrale la care se reduc în ultimă instanţă problemele concrete ale unor domenii capitale ale economiei şi industriei, cum sînt de exemplu planificarea investiţiile, planificarea împrumuturilor şi eşalonarea plăţii dobînzilor, alocarea şi distribuirea resurselor primare (mai ales financiare), etc. Pentru nici una din aceste probleme strategice nu se cunoaşte un *algoritm optim* de rezolvare, ci doar soluţii aproximative. Dacă s-ar cunoaşte algoritmii de soluţionare optimă atunci majoritatea sectoarelor şi proceselor macro- şi micro-economice ar putea fi conduse *în timp real* şi *optim* (!!) cu calculatorul, fără a mai fi necesară prezenţa umană.

Un exemplu cert de domeniu care s-a dezvoltat extraordinar şi în care rolul soft-ului a fost esenţial este chiar domeniul construcţiei de calculatoare, mai ales domeniul proiectării şi asamblării de micro-procesoare. Dacă aţi văzut că schema electronică internă de funcţionare a unui microprocesor din familia Pentium, dacă ar fi desenată clasic, ar ocupa o planşă de dimensiuni 5x5 metri (!), nu mai aveţi cum să vă îndoiţi de faptul că numai un soft de proiectare şi cablare performant mai poate controla şi stăpîni super-complexitatea rezultată. Puţină lume ştie însă că astfel de programe de proiectare performante au putut să apară numai datorită faptului că problema ce stă în spatele funcţionării lor, *problema desenării grafurilor planare,* nu se află pe lista de mai sus a problemelor foarte dificile ale informaticii !

## Probleme nesoluționate încă

Aşa cum s-a putut constata în capitolul anterior, există multe probleme în informatică pentru care încă nu se cunosc soluții eficiente. În continuare vom oferi o listă de probleme nesoluționate încă. De fapt, ele apar mai ales în matematică, fiind cunoscute sub numele de *conjecturi,* şi au toate ca specific un fapt care este de mare interes pentru programatori. Incertitudinea asupra lor ar putea fi definitiv înlăturată nu numai prin demonstrație matematică ci şi cu ajutorul formidabilei puteri de calcul a computerelor. Astfel, fiecare din aceste conjecturi numerice ar putea fi infirmată (concluzia ar fi atunci că *conjectura este falsă*) dacă i s-ar găsi un contraexemplu. Este necesar doar să se găsească un set de numere pentru care propoziția respectivă să fie falsă. Ori, acest efort nu este la îndemîna niciunui matematician dar este posibil pentru un programator înzestrat şi pasionat. El nu are decît să scrie un program eficient şi să pună calculatorul să caute un contra-exemplu.

Atragem atenția asupra unui aspect important. Fiecare problemă conține aceeaşi capcană ca şi în problemele capitolului anterior: algoritmii de căutare a contra-exemplelor pot fi concepuți rapid, relativ simpli şi cu efort de programare redus (de exemplu, prin trei-patru cicluri *for* imbricate sau printr-o soluție gen *back-tracking*) dar ei vor deveni în scurt timp total ineficienți şi vor conduce la programe mari consumatoare de timp. De aceea, vă sugerăm să tratați cu multă atenție problemele din acest capitol. După părerea noastră, abordarea acestui tip de probleme cere din partea programatorului un anumit grad de măiestrie !

Rezolvînd **numai una** dintre ele veți fi recompensați pe măsură: riscați să deveniți celebri !

1. **Conjectura lui Catalan.** Singurele puteri naturale succesive sînt $8=2^3$ şi $9=3^2$.

Observație: într-o exprimare matematică riguroasă, singura soluție în numere naturale *m, n, p, q* a ecuației $n^m+1=p^q$ este *n=2, m=3, p=3 şi q=2*.
Comentariu: avem şirul numerelor naturale *1, 2, 3, 4, 5,...*; încercuind toate puterile de gradul 2: *1, 4, 9, 16, 25,...* apoi toate cele de gradul 3: *1, 8, 27, 64, 125, ...* apoi cele de grad 4, 5, … vom constata că singurele două numere încercuite alăturate sînt 8 şi 9 ! Adică puterile obținute, cu cît sînt mai mari, cu atît au tendința să se "împrăştie" şi să se "distanțeze" unele de altele tot mai tare. În mod misterios, ele nu-şi suportă vecinătatea unele cu altele !

2. **Conjectura cutiei raționale.** Nu se cunoaşte existența unei cutii paralelipipedice avînd lungimile celor trei laturi, ale celor trei diagonale ale fețelor şi a diagonalei principale întregi.

Observație: într-o exprimare matematic riguroasă, nu se cunoaşte să existe trei întregi *a, b, c* astfel încît $a^2+b^2$, $b^2+c^2$ , $c^2+a^2$ şi $a^2+b^2+c^2$ să fie toate patru pătrate perfecte.
Comentariu: în multe subdomenii ale construcțiilor ,de exemplu să ne gîndim la stîlpii de înaltă tensiune ridicați pe vîrfuri înalte de munte şi asamblați în întregime "la fața locului" numai din bare îmbinate cu şuruburi (fără sudură), este de mare interes ca dintr-un număr cît mai mic de subansamble simple (un fel de "cărămizi") să se asambleze obiecte mari cu cît mai multe configurații. Evident, dimensiunile obiectelor rezultate vor avea mărimea ca o combinație întreagă ale dimensiunilor subansamblelor inițiale. După cum rezultă însă din conjectură, se pare că este imposibil să se construiască scheletul întărit (pe diagonale) al unei cutii paralelipipedice din bare de lungimi tipizate. Cel puțin una din diagonale necesită ajustarea lungimii unei bare !

3. **Problema umplerii pătratului unitate.** Întrebare: este posibil ca mulțimea dreptunghiurilor de forma $1/k \times 1/(k+1)$, pentru fiecare k întreg pozitiv, să umple în întregime şi fără suprapuneri pătratul unitate, de latură $1 \times 1$ ?

Observație: este evident că suma infinită a ariilor dreptunghiurilor este egală cu aria pătratului unitate. Avem $\Sigma_{k>0} 1/(k(k+1)) = \Sigma_{k>0} (1/k - 1/(k+1)) = 1$.
Comentariu: aparent, descoperirea dezvoltărilor în serie pare să fi plecat de la unele evidente propietăți geometrice, uşor de sesizat chiar din desene simple în care valorilor numerice li se asociază segmente de lungimi corespunzătoare. Iată însă o surpriză în această situație: suma seriei numerice este evidentă analitic însă reprezentarea geometrică a "fenomenului" este "imposibilă" !

4. **Conjectura fracțiilor egiptene (atribuită lui Erdös şi Graham).** Orice fracție de forma *4/n* se descompune ca sumă de trei fracții egiptene (de forma *1/x*).

Observație: într-o exprimare matematic riguroasă, pentru orice *n* natural există trei valori naturale, nu neapărat distincte, *x, y,* şi *z* astfel încît $4/n=1/x+1/y+1/z$.
Comentariu: este încă un mister motivul pentru care egiptenii preferau descompunerea fracțiilor numai ca sumă de fracții egiptene. Descoperiseră ei această descompunere minimală a fracțiilor de forma $4/n$ ? Dar mai ales, ce procese fizice reale erau astfel mai bine modelate ? Înclinăm să credem că există o legătură între fenomenele fizice ondulatorii, transformata Fourier şi fracțiile egiptene !

5. **Problema punctului rațional.** Există un punct în plan care să se afle la o distanță rațională de fiecare din cele patru vîrfuri ale pătratului unitate ?

Observație: dacă considerăm un pătrat unitate avînd vîrfurile de coordonate *(0,0), (1,0), (0,1)* şi *(1,1)* atunci se cere găsirea unui punct *(x,y)* astfel încît $x^2+y^2$, $(x-1)^2+y^2$, $x^2+(y-1)^2$ şi $(x-1)^2+(y-1)^2$ să fie toate patru pătrate perfecte. Atenție, *x* şi *y* nu este obligatoriu să fie întregi ! Acest fapt ridică foarte serioase probleme la proiectarea unui algoritm de căutare a unui astfel de punct *(x,y)*.
Comentariu: la fel ca şi în cazul cutiei raționale, se pare că există limitări serioase şi neaşteptate în încercarea de optimizare a numărului de subansamble necesare pentru construierea scheletelor sau cadrelor de susținere. Se pare că cele două dimensiuni pe care geometria plană se bazează conduce la o complexitate inerentă neaşteptat de mare !

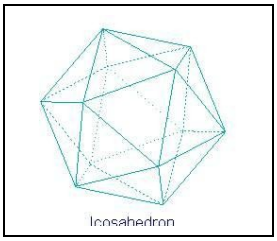
6. **Problema sumei de puteri.** Care este suma seriei de inverse de puteri
$1/1+1/2^3+1/3^3+1/4^3+1/5^3+\ldots$ ?

Observație: se cere să se spună către ce valoare converge seria $\Sigma_{k>0} 1/k^3$ sau $\Sigma_{k>0} k^{-3}$. Se ştie că în cazul în care în locul puterii a 3-ia (cu minus) punem puterea a 2-a (cu minus) seria converge la $\pi^2/6$, în cazul în care în locul puterii a 3-ia punem puterea a 4-a seria converge la $\pi^4/90$.
Comentariu: deşi pare a fi o problemă de analiză matematică pură deoarece ni se cere să găsim expresia sintetică şi nu cea numerică aproximativă a sumei seriei, există însă uluitoare descoperiri asemănătoare ale unor formule de analiză numerică sau chiar dezvoltări în serie (cea mai celebră fiind cea a lui cifrelor hexazecimale ale lui $\pi$) făcute cu ajutorul calculatorului prin calcul simbolic ! Mai multe amănunte găsiți la adresa corespunzătoare de Internet pe care am trecut-o în ultimul capitol.

7. **Problema ecuației diofantice de gradul 5.** Există *a, b, c,* and *d* întregi pozitivi astfel încît $a^5+b^5=c^5+d^5$ ?

Observație: Se cunoaşte că în cazul în care puterea este 3 avem soluția: $1^3+12^3=9^3+10^3$ iar în cazul în care puterea este 4 avem soluția: $133^4+134^4=59^4+158^4$ .
Comentariu: căutarea unor algoritmi generali de rezolvare a ecuațiilor diofantice a condus la importante descoperiri în matematică dar şi în informatică. De exemplu, celebrul matematician *Pierre Fermat,* "stîrnit" fiind de problemele conținute în lucrarea *Arithmetika* a matematicianului antic *Diofant din Alexandria* (de unde şi numele *ecuațiilor diofantice*), a descoperit în 1637 faimoasa sa teoremă: *Ecuația diofantică $x^n+y^n=z^n$ nu admite soluție pentru $n>2$*. Dar tot în aceeaşi perioadă a descoperit şi faptul că cea mai mică soluție a ecuației diofantice $x^2 - 109*y^2 = 1$ este perechea $x=158\ 070\ 671\ 986\ 249$ şi $y= 15\ 140\ 424\ 455\ 100$. Dumneavoastră încercați doar să verificați această soluție fără ajutorul calculatorului şi vă veți putea da seama de performanțele pe care le-a realizat Fermat ! În informatică este acum cunoscut şi demonstrat că este imposibil să se construiască un algoritm general pentru rezolvarea ecuațiilor diofantice !

8. **Problema celor 13 oraşe.** Puteți localiza 13 oraşe pe o planetă sferică astfel încît distanța minimă dintre oricare două dintre ele să fie cît mai mare cu putință ?




Observație: de fapt nu se cunoaşte *cît de mult poate fi mărită* distanța minimală ce se obține dintre cele 78 de distanțe (date de cele $78=C^2_{13}$ de împerechere posibile de oraşe).
Comentariu: dacă s-ar cere localizarea a doar 12 puncte pe sferă, nu este greu de arătat că aşezarea care îndeplineşte condiția cerută este în vîrfurile unui icosaedru (vezi figura alăturată). În acest caz, distanța minimă maximizată este egală cu latura icosaedrului. Este greu de crezut că în cazul descoperirii aşezării a 13 puncte pe sferă se poate porni tocmai de la

icosaedru ! Evident că în rezolvarea aplicativ-practică a acestui tip de probleme nesoluționate geometric pînă în prezent rolul programatorului poate fi capital. La ora actuală pentru astfel de situații se oferă soluții aproximative. Acestea constau din algoritmi care încearcă să aproximeze cît mai exact soluția optimă într-un timp rezonabil de scurt. Evident că în aceste condiții algoritmii de căutare exhaustivă (gen back-tracking) sînt cu totul excluşi !

9. **Conjectura lui Collatz.** Se pleacă de la un *n* întreg pozitiv. Dacă *n* este par se împarte la doi; dacă *n* este impar se înmulțește cu trei şi i se adună unu. Repetînd în mod corespunzător doar aceşti doi paşi se va ajunge întotdeauna la *1* indiferent de la ce valoare *n* se pornește ?

Observație: de exemplu, pornind de la *n=6* obținem în 8 paşi şirul valorilor: *6, 3, 10, 5, 16, 8, 4, 2, 1*.
Comentariu: valoarea finală *1* este ca o "gaură neagră" care absoarbe în final şirul obținut. "Raza" de-a lungul căreia are loc "căderea" în gaura neagră *1* este dată mereu de şirul puterilor lui 2: *2, 4, 8, 16, 32, 64, ...* cu ultima valoare de forma *3k+1*, adică *4, 16, 64, 256, ....* Se pare că valorile obținute prin cele două operații nu pot "să nu dea" nicicum peste acest şir care le va face apoi să "cadă în gaura neagră" 1!

10. **Problema înscrierii pătratului.** Dîndu-se o curbă simplă închisă în plan, vom putea întotdeauna găsi patru puncte pe curbă care pot să constituie vîrfurile unui pătrat ?

Observație: în cazul curbelor închise regulate (ce au axe de simetrie: cerc, elipsă, ovoid) nu este greu de arătat prin construire efectivă că există un pătrat ce se "sprijină" pe curbă. Pare însă de nedovedit acelaşi fapt în cazul unor curbe închise foarte neregulate ! Găsirea celor patru puncte, într-o astfel de situație, este de neimaginat fără ajutorul calculatorului !
Comentariu: o consecință surprinzătoare a acestei conjecturi este faptul că pe orice curbă de nivel (curbă din teren care unește punctele aflate toate la aceaşi altitudine) am putea găsi patru puncte de sprijin pentru o platformă pătrată (un fel de masă) perfect orizontală, de mărime corespunzătoare. Acest fapt ar putea să explice ampla răspîndire a meselor cu patru picioare (!?) în detrimentul celor cu trei: dacă îi cauți poziția, cu siguranță o vei găsi şi o vei putea aşeza pe toate cele patru picioare, astfel masa cu patru picioare va oferi o perfectă stabilitate şi va sta perfect orizontală, pe cînd cea cu trei picioare deşi stă acolo unde o pui din prima (chiar şi înclinată) nu oferă aceeaşi stabilitate.

În speranța că am reuşit să vă stîrnim interesul pentru astfel de probleme nesoluționate încă şi care sînt grupate pe Internet în liste cuprinzînd zeci de astfel de exemple (vezi adresa oferită în ultimul capitol), încheiem acest capitol cu următoarea constatare: descoperirile deosebite din matematica actuală au efecte rapide şi importante nu numai în matematică ci şi în informatică. Să oferim doar un singur exemplu de mare interes actual: algoritmii de încriptare/decriptare cu cheie publică, atît de folosiți în comunicația pe Internet, se bazează în întregime pe proprietățile matematice ale divizibilității numerelor prime.

Ceea ce este interesant şi chiar senzațional este faptul că în informatică nevoia de programe performante a condus la implementarea unor algoritmi care se bazează pe cele mai noi descoperiri din matematică, chiar dacă acestea sînt încă în stadiul de conjecturi! De exemplu, pentru acelaşi domeniu al criptării cu cheie publică există deja algoritmi de primalitate senzațional de performanți care se bazează pe Ipoteza (conjectura) lui Riemman. (Mai multe amănunte puteți găsi la adresele de Internet pe care le oferim în ultimul capitol.)

Este acest fapt legitim ? Ce încredere putem avea în astfel de programe ? După părerea noastră putem acorda o totală încredere acestor algoritmi dar numai în limitele "orizontului" atins de programele de verificare a conjecturii folosite. Dacă programul de verificare a verificat conjectura numerică pe intervalul 1- $10^{30}$ atunci orizontul ei de valabilitate este $10^{30}$. Domeniile numerice pe care le pot acoperi calculatoarele actuale sînt oricum foarte mari şi implicit oferă o precizie suficientă pentru cele mai multe calcule cu valori extrase din realitatea fizică.

## Probleme insolvabile algoritmic

Am introdus acest capitol special din două motive. Primul motiv, pentru a trezi interesul şi pasiunea pentru informatică celor care pot acum să vadă cît de deosebite sînt descoperirile şi rezultatele din acest domeniu. Al doilea motiv, pentru ai pune în gardă pe cei care, în entuziasmul lor exagerat, îşi închipuie că pot programa calculatorul să facă orice treabă sau să rezolve orice problemă. Aşa cum am văzut şi în capitolul ce tratează despre problemele dificile ale informaticii, enunţurile problemelor foarte dificile sau chiar insolvabile sînt foarte simple şi pot uşor constitui o capcană pentru necunoscători.

În continuare vom oferi spre edificare doar cîteva exemple, urmînd ca prin studiul *Complexităţii algoritmilor şi a dificultăţii problemelor* să se aprofundeze acest domeniu fascinant dar atît de uşor de confundat (poţi să dai de aceste probleme chiar şi din greşeală !?) şi care este păcat să fie tratat într-un mod superficial.

1. **Problema Stopului. Nu există un algoritm universal valabil** prin care să se poată decide dacă execuţia oricărui algoritm se opreşte vreodată sau nu.

Comentariu: acesta este cel dintîi şi cel mai celebru exemplu de problemă insolvabilă. Demonstraţia riguroasă a acestui fapt a fost dată pentru prima dată în 1936 de inventatorul calculatorului actual matematicianul englez Alan Mathison Turing. Odată existînd această demonstraţie, multe din următoarele probleme insolvabile algoritmic s-au redus la aceasta. Implicaţiile practice, teoretice şi filozofice ale problemei Stopului sînt foarte importante atît pentru informatică cît şi pentru matematică. Astfel, două consecinţe strategice ale problemei Stopului sînt: 1. nu poate exista un calculator oricît de puternic cu ajutorul căruia să se poată decide asupra comportamentului viitor al oricărui alt calculator de pe glob; 2. nu poate să existe în matematică o metodă generală de demonstrare inductivă-logică a propoziţiilor matematice (se închide în acest fel o mai veche căutare a matematicienilor şi logicienilor cunoscută sub numele de *Entscheidungs Problem* sau *Problema deciziei*).

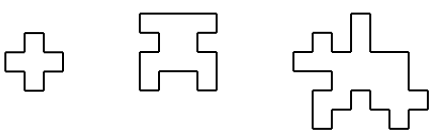
2. **Problema ecuaţiilor diofantice. Nu există o metodă generală (un algoritm)** de aflare a soluţiilor întregi ale unui sistem de ecuaţii diofantice.

Comentariu: sistemele de ecuaţii diofantice sînt sistemele de ecuaţii algebrice de mai multe variabile cu coeficienţi întregi şi cărora li se caută soluţii întregi. De exemplu, a fost nevoie de ajutorul calculatorului pentru a se descoperi cea mai mică soluţie a ecuaţiei diofantice $p^4+q^4+r^4=s^4$ şi care este cvadrupletul $p=95600,\ q=217519,\ r=414560,\ s=422461$ (infirmîndu-se în acest fel "conjectura" lui *Leonard Euler* care în 1796 a presupus că această ecuaţie diofantică nu are soluţii întregi). Această problemă ce cere o metodă generală de rezolvare a ecuaţiilor diofantice este cunoscută sub denumirea de *Problema a 10-a a lui Hilbert.*

3. **Problema acoperirii planului (Problema pavajului** sau **Problema croirii).** Fiind dată o mulţime de forme poligonale, **nu există o metodă generală (un algoritm)** care să decidă dacă cu aceste forme este posibilă acoperirea completă a planului (fără suprapuneri şi goluri).

Comentariu: în practică este mult mai importantă problema croirii care cere să se decupeze *fără pierderi* un set cît mai mare de forme date (croiuri) dintr-o bucată iniţială de material oricît de mare. Este deasemenea demonstrat că problema rămîne insolvabilă algoritmic chiar şi atunci cînd formele poligonale sînt reduse la *poliomine* (un fel de "mozaicuri") care se formează doar pe o reţea rectangulară caroiată. Iată cîteva exemple de mulţimi formate dintr-o singură poliomină şi, alăturat, răspunsul la întrebarea dacă cu ele se poate acoperi planul sau nu:

DA NU DA



4. **Problema şirurilor lui Post.** Se dau două mulţimi egale de şiruri finite de simboluri ce sînt împerecheate astfel: un şir dintr-o mulţime cu şirul corespunzător din a doua mulţime. **Nu există**

**un algoritm general** prin care să se decidă dacă există o ordine de concatenare a şirurilor (simultan din cele două mulţimi) astfel încît cele două şiruri lungi pereche rezultate să fie identice.

Comentariu: de exemplu, fie A={ 101, 010, 00 } şi B={ 010, 10, 001 } cele două mulţimi de şiruri de simboluri (pentru uşurinţă au fost alese simbolurile binare 1 şi 0). Perechile corespunzătoare de şiruri sînt 1.(101,010), 2.(010,10) şi 3.(00,001). Observăm că şirurile pereche pot avea lungimi diferite (ca în perechile 2 şi 3). În continuare, pentru a vedea cum se procedează, cele două şiruri pereche rezultante prin concatenare le vom scrie unul deasupra celuilalt sesizînd cum avansează procesul de egalizare a lor. Punctele sînt intercalate doar pentru a evidenţia perechile, ele nu contribuie la egalitate, iar comentariile ne aparţin:

| | | | |
|---|---|---|---|
| 00.<br>001. | Concatenarea poate începe doar cu perechea a 3-a,00 de "sus" ⊂ 001 de "jos" | 00.101.<br>001.010. | Obligatoriu urmează perechea 1-a singura care începe cu 1 "sus". |
| 00.101.00.<br>001.010.001. | Dacă am continua cu perechea a 3-a … | 00.101.010<br>001.010.10 | … nu s-ar obţine rezultatul final oferit de perechea 2-a ! |

5. **Problema cuvintelor "egale".** Se dă un anumit număr de "egalităţi" între cuvinte. Bazîndu-ne pe aceste "egalităţi" se pot obţine unele noi substituind apariţiile cuvintelor dintr-o parte a egalului cu cele din cealaltă parte. **Nu există un algoritm general** de a decide dacă un cuvînt oarecare A poate fi "egal" cu un altul B.

Comentariu: de exemplu, fie următoarele cinci egalităţi (citiţi-le în limba engleză) EAT=AT, ATE=A, LATER=LOW, PAN=PILLOW şi CARP=ME. Este CATERPILLAR egal cu MAN ? Iată şirul egalităţilor iterate care ne poate oferi răspunsul: C<u>ATE</u>RPILLAR = C***A***RPILL<u>A</u>R =CARPIL<u>***ATE***</u>R =CAR<u>PIL</u>***LOW***= CAR***PAN***= ***ME***<u>A</u>N= ME***ATE***N= M***AT***EN= M***A***N.
Dar de la CARPET putem ajunge la MEAT ? Întrucît se vede că numărul total de A-uri plus W-uri şi M-uri nu se poate modifica prin nici o substituţie şi întrucît CARPET are un A (adică numărul asociat este 1) iar MEAT are un A şi un M (deci 2), rezultă că această egalitate nu este permisă.
Mai mult, se ştie că există liste particulare de cuvinte pentru care nu poate exista un algoritm ce decide dacă două cuvinte sînt egale sau nu. Iată o astfel de listă de şapte egalităţi: AH=HA, OH=HO, AT=TA, OT=TO, TAI=IT, HOI=IH şi THAT=ITHT.

Numărul problemelor cunoscute ca fiind insolvabile algoritmic este destul de mare. Cele mai multe probleme provin din matematică, subdomeniul matematicii care studiază aceste probleme se numeşte *Matematica nerecursivă.* De aceea ele pot fi întîlnite mai ales sub numele de *probleme nedecidabile* sau *probleme nerecursive,* în enunţul lor cuvîntul *algoritm* fiind înlocuit mai ales cu cuvintele *metodă generală.*

Studierea acestui domeniu a creat condiţii pentru apariţia de noi direcţii de cercetare prin care se încearcă explicarea raţionamentelor matematice ba chiar se încearcă descoperirea limitelor raţiunii umane în general. Unii oameni de ştiinţă contemporani, cum este celebrul matematician-fizician englez *Roger Penrose*, depun eforturi mari pentru a oferi o demonstraţie matematică riguroasă pentru ipoteza că, în cele din urmă şi în esenţă, *raţionamentele umane nu sînt algoritmice*, nici măcar cele matematice. După părera lui Penrose mintea umană nu poate fi asimilată cu un calculator ci este mai mult decît atît şi nu vor putea exista vreodată calculatoare sau roboţi mai inteligenţi decît oamenii! În ultimul capitol oferim titlurile cărţilor recent apărute ce tratează despre acest fascinant subiect .

## Noțiuni aprofundate de programare

# Metode şi strategii de proiectare a algoritmilor (alias tehnici de programare)

În rezolvarea sa cu ajutorul calculatorului orice problemă trece prin trei etape obligatorii: *Analiza problemei, Proiectarea algoritmului de soluționare* şi *Implementarea algoritmului într-un program pe calculator*. În ultima etapă, sub acelaşi nume, au fost incluse în plus două subetape cunoscute sub numele de *Testarea* şi *Întreținerea programului.* Aceste subetape nu lipsesc din "ciclul de viață" a oricărui produs-program ce "se respectă" dar , pentru simplificare, în continuare ne vom referi doar la cele trei mari etape..

Dacă etapa *implementării* algoritmului într-un program executabil este o etapă exclusiv practică, realizată "în fața calculatorului", celelalte două etape au un caracter teoretic pronunțat. În consecință, primele două etape sînt caracterizate de un anumit grad de abstractizare. Din punct de vedere practic şi în ultimă instanță criteriul decisiv ce conferă succesul rezolvării problemei este dat de calitatea implementării propriuzise. Mai precis, succesul soluționării este dat de performanțele programului: utilitate, viteză, fiabilitate, manevrabilitate, lizibilitate, etc. Este imatură şi neprofesională "strategia" programatorilor începători care neglijînd primele două etape sar direct la a treia, fugind de analiză şi de componenta abstractă a efortului de soluționare. Ei oferă cu toții aceeaşi justificare: "Eu nu vreau să mai pierd vremea cu …, am să fac programul cum ştiu eu. Pînă cînd nu o să facă cineva altul mai bun decît al meu, pînă atunci…nu am cu cine sta de vorbă !".

Este adevărat că ultima etapă în rezolvarea unei probleme – implementarea – este într-adevăr decisivă şi doveditoare, dar primele două etape au o importanță capitală. Ele sînt singurele ce pot oferi răspunsuri la următoarele întrebări dificile: *Avem certitudinea că soluția găsită este corectă ? Avem certitudinea că problema este complet rezolvată ? Cît de eficientă este soluția găsită ? Cît de departe este soluția aleasă de o soluție optimă ?*

Să menționăm în plus că literatura de specialitate conține un număr impresionant de probleme "capcană" pentru începători şi nu numai. Ele sînt toate inspirate din realitatea imediată dar pentru fiecare dintre ele nu se cunosc soluții eficiente în toată literatura de profil. Există printre ele chiar unele probleme extrem de dificile pentru care **s-a demonstrat riguros** că nu admit soluție cu ajutorul calculatorului. (Mai precis, s-a demonstrat că ele nu admit soluție prin metode algoritmice, în spiritul tezei Turing-Church). Cîți dintre programatorii începători n-ar fi surprinşi să afle că problema "atît de simplă" (ca enunț) a cărei soluționare tocmai au abandonat-o este de fapt o problemă dovedită ca fiind intratabilă sau chiar insolvabilă algoritmic ? Partea proastă a lucrurilor este că, aşa cum ciupercile otrăvite nu pot fi cu uşurință deosebite de cele comestibile, tot astfel problemele netratabile pot fi cu uşurință confundate cu nişte probleme uşoare la o privire rapidă şi lipsită de experiență.

Să înțelegem mai întîi care este "cheia" ce conduce la răspunsuri pentru întrebările de mai sus iar apoi vom trece la prezentarea metodelor clasice de proiectare a soluțiilor. Aceste metode de proiectare a algoritmilor-soluție sînt cunoscute în literatura de specialitate sub numele de *tehnici de programare* şi sînt considerate metode sau instrumente soft eficiente şi cu arie largă de acțiune.

Dacă ar fi să sintetizăm în cîte un cuvînt efortul asupra căruia se concentrează fiecare din primele două etape – *analiza* şi *proiectarea* – acestea ar fi: *corectitudine* şi *eficiență*. Etapa de analiză este singura care permite *dovedirea cu argumente riguroase* a corectitudinii soluției, iar etapa de proiectare este singura care poate *oferi argumente precise* în favoarea eficienței soluției propuse.

În general problemele de informatică au în forma lor inițială sau în enunț o caracteristică pragmatică. Ele sînt foarte ancorate în realitatea imediată şi aceasta le conferă o anumită asemănare. Totuşi ele au în forma inițială un grad mare de *eterogenitate*, *diversitate* şi *lipsă de rigoare.* Fiecare dintre aceste atribute "negative" este un obstacol major pentru demonstrarea corectitudinii soluției. Rolul esențial al etapei de analiză este deci acela de a transfera problema "de pe nisipurile mişcătoare" ale realității imediate de unde ea provine într-un plan abstract, adică de *a o modela*. Acest "univers paralel" este dotat cu mai multă rigoare şi disciplină internă, avînd legi precise, şi poate oferi instrumentele logice şi formale necesare pentru demonstrarea riguroasă a corectitudinii soluției problemei.

Planul abstract în care sînt "transportate" toate problemele este planul sau *universul obiectelor matematice.* Acest *univers al matematicii* este unanim acceptat (de ce ?!) iar corespondentul problemei în acest plan va fi *modelul matematic abstract* asociat problemei. Demonstrarea corectitudinii proprietăților ce leagă obiectele universului matematic a fost şi este sarcina matematicienilor. Celui ce analizează problema din punct de vedere informatic îi revine sarcina (nu tocmai uşoară) de a dovedi printr-o **demonstrație constructivă** că există o *corespondență precisă (bijectivă)* între părțile

componente ale problemei reale, "dezasamblată" în timpul analizei, şi părţile componente ale modelului abstract asociat. Odată descoperită, formulată precis şi dovedită, această "perfectă oglindire" a problemei reale în planul obiectelor matematice oferă certitudinea că toate proprietăţile şi legăturile ce există între subansamblele modelului abstract se vor regăsii precis (prin reflectare) între părţile interne ale problemei reale, şi invers. Atunci, soluţiei abstracte descoperită cu ajutorul modelului matematic abstract îi va corespunde o soluţie reală concretizată printr-un algoritm ce poate fi implementat într-un program executabil.

Aceasta este calea generală de rezolvare a problemelor şi orice poate verifica. Ca şi exerciţiu, să se demonstreze corectitudinea (să se aducă argumente precise, clare şi convingătoare în favoarea *corectitudinii*) metodei de extragere a radicalului învăţată încă din şcoala primară sau a algoritmului lui Euclid de determinare a celui mai mare divizor comun a două numere prin împărţiri întregi repetate. Argumentele elevilor de forma: "Este corect pentru că aşa ne-a învăţat doamna profesoară!" sau "Este corect pentru că aşa face toată lumea !" sînt "normale" atît timp cît nu li se oferă o argumentaţie matematică riguroasă.

Ideea centrală a etapei a doua – proiectarea uni algoritm de soluţionare eficient poate fi formulată astfel: din studiul *proprietăţilor* şi *limitelor* modelului matematic abstract asociat problemei se deduc *limitele inferioare ale complexităţii minimale ("efortului minimal obligatoriu")* inerente **oricărui** algoritm ce va soluţiona problema în cauză. *Complexitatea internă a modelului abstract* şi *complexitatea soluţiei abstracte* se va reflecta imediat asupra *complexităţii reale a algoritmului*, adică asupra *eficienţei*, de soluţionare a problemei. Acest fapt permite *prognosticarea* încă din această fază – faza de proiectare a algoritmului de soluţionare – a eficienţei practice, măsurabilă ca *durată de execuţie*, a programului.

Această corespondenţă exactă între complexitatea modelului abstract şi complexitatea algoritmului de soluţionare oferă cheia unor demonstraţii riguroase a imposibilităţii existenţei soluţiei prin metode algoritmice pentru o listă întreagă de probleme (cum ar fi de exemplu Problema a 10-a a lui Hilbert, formulată încă din 1900).

Detailînd cele prezentate deja, vom construi în continuare cadrul teoretic general pentru înţelegerea strategiilor de proiectare a algoritmilor.

Creşterea impresionantă a puterii de calcul a calculatoarelor i-a "obligat" pe informaticienii ultimilor treizeci de ani să nu se mai eschiveze de la abordarea problemelor dificile cu caracter algoritmic din diverse domenii care au intrat în atenţia matematicienilor încă de la începutul acestui secol. De altfel, astfel de probleme cu soluţii algoritmice nu constituiau neapărat o noutate pentru matematicienii începutului de secolul. Încă de pe vremea lui Newton matematicienii şi-au pus, de exemplu, problema descoperirii unor metode precise (adică algoritmi!) de determinare în paşi de aproximare succesivă a soluţiei unei ecuaţii ce nu poate fi rezolvată prin radicali. Dar "boom-ul" dezvoltării tehnicii de calcul din a doua jumătate a secolului a creat posibilitatea abordării unor probleme cheie pentru anumite domenii strategice (de exemplu, controlul şi dirijarea sateliţilor pe orbită, probleme de planificare sau optimizare în economie, etc.) care se reduc în fapt la soluţionarea eficientă a unor probleme de optimizare matematică prin metode iterative (algoritmi).

Spre deosebire de aceste probleme a căror succes în soluţionare a fost total şi cu consecinţele ce se văd, există însă o serie de probleme dificile inspirate din realitate care se cer imperios rezolvate eficient cu ajutorul calculatorului.

Principală caracteristică a acestora este că, datorită generalităţii lor sau datorită dificultăţii "ascunse", în literatura de specialitate nu există metode iterative eficiente de rezolvare a lor şi nu se ştie dacă ele admit astfel de soluţii. Singurul fapt ce poate fi stabilit dinainte în cazul soluţionării unor astfel de probleme este "spaţiul" în care soluţia trebuie căutată. Ceea ce trebuie atunci construită este o *strategie* corectă şi cît mai generală de căutare a soluţiei (soluţiilor) în acel spaţiu de căutare a soluţiilor.

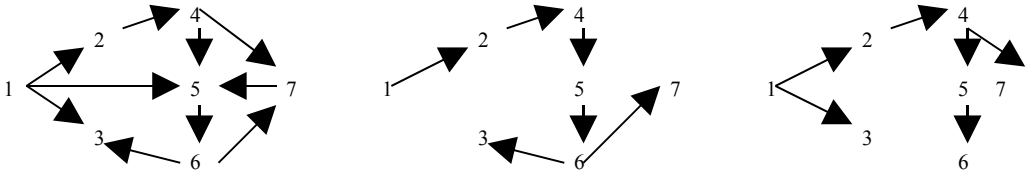
Exemplu concret: există o clasă întreagă de probleme ce cer implicit să se genereze toate obiectele unei mulţimi (cum ar fi problema generării tuturor permutărilor unei mulţimi cu $n$ elemente). În acest caz este cunoscută dinainte proprietatea ce trebuie să o îndeplinească fiecare soluţie ca să fie un obiect al spaţiului de căutare a soluţiilor. Efortul de soluţionare va fi redus atunci la aflarea, căutarea sau generarea pe baza proprietăţii respective a *tuturor* obiectelor posibile, fără însă a lăsa vreunul pe dinafară.

Modelul matematic abstract cel mai general care permite modelarea acestui tip de probleme este *graful*. Un graf este un obiect matematic riguros care, simplificat, poate fi privit ca fiind o diagramă formată din noduri unite prin săgeţi (muchii). De exemplu, orice hartă feroviară sau rutieră poate fi privită ca un graf cu mulţimea nodurilor formată din localităţi iar mulţimea muchiilor formată din rutele de transport directe dintre localităţile respective. Graful permite descrierea legăturilor şi a relaţiilor ce există între diferite obiecte abstracte reprezentate prin noduri. Experienţa arată că acest *model matematic abstract* este *cel mai general* şi cel mai potrivit pentru descrierea unui spaţiu de

căutare a soluțiilor unei probleme. În cazul spațiului de căutare, nodurile sînt soluțiile posibile (ipotetice). Două noduri în graf vor fi unite prin săgeți (muchii) dacă cele două soluții posibile au în comun o aceeaşi proprietate. Muchiile grafului sînt "punțile" ce vor permite algoritmului trecerea de la un nod la altul, de la o soluție ipotetică la alta, în timpul procesului de căutare a soluției (sau a tuturor soluțiilor). Rezultă că strategiile cele mai generale de căutare a soluției (soluțiilor) pe modelul abstract asociat problemei sînt reductibile la *metodele generale de traversare a unui graf*.

Ordinea de traversare a grafului determină precis *arborele de traversare* a grafului. Acest arbore este de fapt un subgraf particular al grafului inițial, avînd acelaşi număr de noduri şi ca rădăcină vîrful inițial de pornire. Cele două metode clasice de traversare a unui graf (căutare într-un graf) poartă în literatura de specialitate numele: *BreathFirstSearch* (BFS) şi *DepthFirstSearch* (DFS), respectiv *Traversarea în lățime* (sau traversarea pe nivele) şi *Traversarea în adîncime* (traversarea "labirintică") a grafului. Ambele metode stau la baza celei mai cunoscute strategii de proiectare a algoritmilor (impropriu denumită tehnică de programare): *BackTracking* respectiv *căutare (traversare) în spațiul de căutare a soluțiilor (a grafului) cu revenire pe "urma" lăsată*.

Iată un exemplu de graf (7 noduri şi 10 arce-săgeți) şi ordinea sa de traversare prin cele două metode:




Ordinea de parcurgere a celor 7 vîrfuri ale grafului, ținînd cont şi de sensul dat de săgeți, este în cazul DFS (în adîncime): 1,2,4,5,6,3,7 aşa cum se vede din arborele parcurgerii în adîncime. Din fiecare nod se continuă cu nodul (nevizitat încă) dat de prima alegere posibilă: de exemplu, din 4 se continuă cu 5 (ales în favoarea lui 7). Se poate observa cum din nodul 3, nemaiexistînd continuare, are loc o revenire pe "pista lăsată" pînă în nodul 6 de unde se continuă parcurgerea în adîncime cu prima alegere posibilă. În cazul BFS (în lățime) ordinea de traversare este: 1,2,3,4,5,7,6 aşa cum se poate vedea în arborele parcurgerii în lățime. În această situație, dintr-un nod sînt vizitați toți vecinii (nevizitați încă), iar apoi se face repetă acelaşi lucru pentru fiecare nod vecin, în ordinea vizitării. Se observă cum nodul 7 este vizitat înaintea nodului 6, fiind vecin cu nodul 4. (De fapt, aceasta se explică prin faptul că distanța de la 1 la 7 este mai mică cu o unitate decît distanța de la 1 la 6.) Putem spune că în cazul traversării în lățime ordinea de traversare este dată de depărtarea nodurilor față de nodul de start.

Iată cum arată procedura generală *DepthFirstSearch (DFS)* de traversare a unui graf descrisă în pseudo-cod în varianta recursivă:

**Procedura DFS(v:nod);**
Vizitează v;
Marchează v; // v devine un nod vizitat //
Cît timp (există w nemarcat nod adiacent lui v) execută **DFS**(w);

***Să nu uităm că această procedură poate fi privită ca "scheletul" pe care se construieşte orice procedură backtracking recursivă.***

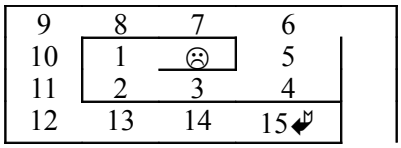
## BackTracking.

Pentru a preciza mai exact în ce constă această metodă, vom relua pe un exemplu concret cele spuse deja. Avem următoarea problemă: se cere generarea tuturor permutărilor unei mulțimi de $n$ elemente ce nu conțin elementul $x$ (dat dinainte) pe primele două poziții. Conform celor afirmate, este suficient să "construim" modelul abstract - graful - (mai precis arborele) tuturor permutărilor celor $n$ elemente. Apoi, printr-o parcurgere exhaustivă a nodurilor sale, prin una din metodele BFS sau DFS, să păstrăm numai acele noduri ce verifică în momentul "vizitării" condiția impusă (lipsa lui $x$ de pe primele două poziții).

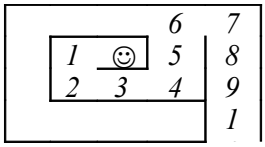
Observăm că această metodă necesită folosirea în scopul memorării dinamice a drumului parcurs (în timpul căutării soluției) a mecanismului de *stivă*, fapt sugerat chiar de numele ei: tracking, adică înregistrarea pistei parcurse. Acest mecanism de stivă, care permite atît memorarea pistei cît şi

revenirea – back-tracking-ul, este unul din mecanismele de bază ce este folosit pe scară largă în procedurile de gestiune dinamică a datelor în memorie. În plus, există unele cazuri particulare de probleme în care soluția căutată se obține în final prin "vărsarea" întregului conținut al stivei şi nu doar prin "nodul" ultim vizitat, aflat în vîrful stivei.

Exemplul cel mai potrivit de problemă ce necesită o strategie de rezolvare backtracking este *Problema Labirintului*: se cere să se indice, pentru o configurație labirintică dată, traseul ce conduce către ieşirea din labirint. Iată un exemplu sugestiv:




Observați cum, după 15 paşi, este necesară o revenire (backtracking) pînă la căsuța 6, de unde se continuă pe o altă pistă. "Pista falsă" a fost memorată în stivă, element cu element, iar revenirea se va realiza prin eliminarea din stivă tot element cu element. Cînd în vîrful stivei reapare căsuța cu numărul 6, stiva începe din nou să crească memorînd elementele noului drum. În final stiva conține în întregime soluția: drumul corect către ieşirea din labirint.




În consecință, indiferent de forma particulară ce o poate lua sau de modul în care este "citită" în final soluția, esențialul constă în faptul că *backtracking-ul este o metodă de programare ce conține obligatoriu gestiune de stivă*. Lipsa instrucțiunilor, explicite sau "transparente", de gestionare a stivei într-un program (de exemplu, lipsa apelului recursiv), este un indiciu sigur de recunoaştere a faptului că acel algoritm nu foloseşte metoda sau strategia de rezolvare BackTracking.

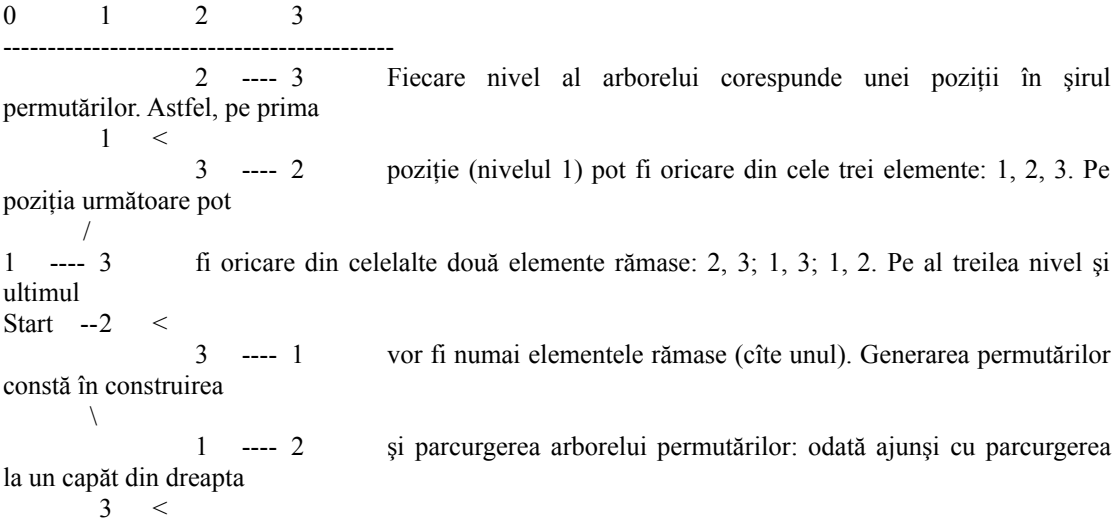
Tot o metodă back-tracking este şi metoda de programare cunoscută sub numele *programare recursivă.* Ea este mai utilizată decît metoda clasică BackTracking, fiind mai economicoasă din punctul de vedere al minimizării efortului de programare. Această metodă se reduce la construirea, în mod transparent pentru programator, a arborelui apelurilor recursive, traversarea acestuia prin apelarea recursivă (repetată) şi efectuarea acțiunilor corespunzătoare în momentul "vizitării" fiecărui nod al arborelui. Apelarea recursivă constituie "motorul vehiculului" de traversare şi are doar rolul de a permite traversarea arborelui. Gestionarea stivei apelurilor recursive şi revenirea - back-tracking-ul rămîne în sarcina mediului de programare folosit şi se efectuează într-un mod mascat pentru programator. Din acest punct de vedere, programatorului îi revine sarcina scrierii corecte a instrucțiunii de apel recursiv şi a instrucțiunii ce "scurt-circuitează" bucla infinită a apelurilor recursive. Singurele instrucțiuni care "fac treabă", în sensul rezolvării propriuzise a problemei respective, sînt cele cuprinse în corpul procedurii.

De exemplu, iată cum arată în limbajul de programare Pascal procedura generală de generare a permutărilor în varianta recursivă şi arborele de generare a permutărilor mulțimii {1,2,3} (*n=3*), pe nivele:

```
Procedure Permut(k:byte;s:string);          { k - nivelul în arbore, s - şirul}
Var i:byte;tmp:char;
Begin
    If k=n then begin                       { scurt-circuitarea recursivității}
    For i:=1 to n do Write(s[i]);           { prin afişarea permutării        }
       Write(';');                          { urmată de un punct-virgulă      }
    end else
        For i:=k to n do begin              { singurele instrucțiuni "ce fac treabă" }
        tmp:=s[i];s[i]:=s[k];s[k]:=tmp;     { sînt for-ul şi cele trei atribuiri }
            Permut(k+1,s);                  { apelul recursiv ce permite parcugerea  }
        end;                                { arbor. de generare a celor n! permutări}
End;
```

**Nivelele arborelui** (răsturnat pe orizontală)

---




Fiecare nivel al arborelui corespunde unei poziții în şirul permutărilor. Astfel, pe prima poziție (nivelul 1) pot fi oricare din cele trei elemente: 1, 2, 3. Pe poziția următoare pot fi oricare din celelalte două elemente rămase: 2, 3; 1, 3; 1, 2. Pe al treilea nivel şi ultimul vor fi numai elementele rămase (cîte unul). Generarea permutărilor constă în construirea şi parcurgerea arborelui permutărilor: odată ajunşi cu parcurgerea la un capăt din dreapta vom afişa de fiecare dată "drumul" de la "rădăcină" la "frunza" terminală.

Observăm că arborele permutărilor este identic cu arborele apelurilor recursive şi că controlul şi gestiunea stivei se face automat, transparent față de programator. Instrucțiunilor de control (din background) le revine sarcina de a păstra şi de a memora, de la un apel recursiv la altul, string-ul *s* ce conține permutările. Deşi această procedură recursiv de generare a permutărilor pare o variantă de procedură simplă din punctul de vedere al programatorului, în realitate, ea conține într-un mod ascuns efortul de gestionare a stivei: încărcarea-descărcarea stringului *s* şi a întregului *k*. Acest efort este preluat în întregime de instrucțiunile incluse automat de mediul de programare pentru realizarea recursivității.

Avantajul metodei back-tracking este faptul că efortul programatorului se reduce la doar trei sarcini:

1. "construirea" grafului particular de căutare a soluțiilor
2. adaptarea corespunzătoare a uneia din metodele generale de traversare-vizitare a grafului în situația respectivă (de exemplu, prin apel recursiv)
3. adăugarea instrucțiunilor "ce fac treabă" care, fiind apelate în mod repetat în timpul vizitării nodurilor (grafului soluțiilor posibile), rezolvă gradat problema, găsind sau construind soluția.

Acțiunea de revenire ce dă numele metodei, *backtracking - revenire pe "pista lăsată"*, este inclusă şi este efectuată de subalgoritmul de traversare a grafului soluțiilor posibile. Acest subalgoritm are un caracter general şi face parte din "zestrea" oricărui programator. În cazul particular în care graful soluțiilor este arbore, atunci se poate aplica întotdeauna cu succes metoda programării recursive care conduce la un cod-program redus şi compact.

Prezentăm din nou procedura generală *DepthFirstSearch (DFS)* de traversare a unui graf în varianta recursivă (ce "construieşte" de fapt arborele de traversare a grafului având ca rădăcină nodul de pornire) pentru a pune în evidență cele spuse.

```
Procedura DFS(v:nod);
  Vizitează v;                                  { aici vor fi acele instrucțiuni "care fac treabă" }
  Marchează v;   // v devine un nod vizitat //  { poate să lipsească în anumite implementări    }
Cît timp (există w nemarcat nod adiacent lui v)
        execută DFS(w);                         { apelul recursiv este "motorul vehiculului" }
                                        { ce permite  parcurgerea  grafului şi gestiunea stivei de revenire }
```

Există situații în care, la unele probleme, putem întîlni soluții tip-backtracking fără însă a se putea sesiza la prima vedere prezența grafului de căutare asociat şi acțiunea de traversare a acestuia, ci doar prezența stivei. O privire mai atentă însă va conduce obligatoriu la descoperirea arborelui de căutare pe graful soluțiilor, chiar dacă el există doar într-o formă mascată. Acest fapt este inevitabil şi constituie esența metodei – căutare (generare) cu revenire pe pista lăsată.

Back-tracking-ul, metodă generală şi cu o largă aplicabilitate, fiind reductibilă în ultimă instanță la traversarea spațiului -grafului de căutare- a soluțiilor, are marele avantaj că determină cu certitudine toate soluțiile posibile, cu condiția ca graful asociat de căutare a soluțiilor să fie corect. Dar

ea are marele dezavantaj că necesită un timp de execuție direct proporțional cu numărul nodurilor grafului de căutare asociat (sau numărul cazurilor posibile). În cele mai multe cazuri acest număr este exponențial ($e^n$) sau chiar mai mare, factorial ($n!$), unde $n$ este dimensiunea vectorului datelor de intrare. Acest fapt conduce la o durată de execuție de mărime astronomică făcînd într-un astfel de caz algoritmul complet inutilizabil, chiar dacă el este corect teoretic. (De exemplu, dacă soluționarea problemei ar necesita generarea tuturor celor *100!* permutări ($n=100$), timpul de execuție al algoritmului depăşeşte orice imaginație.) În astfel de situații, în care dimensiunea spațiului de căutare-generare a soluțiilor are o astfel de dependență în funcție de $n$ (fiind o funcție de ordin mai mare decît funcția polinomială), este absolut necesară îmbunătățirea acestei metode sau înlocuirea ei. Nu este însă necesară (şi de multe ori nici nu este posibilă!) abandonarea modelului abstract asociat - graful soluțiilor posibile, cu calitățile şi proprietățile sale certe - ci este necesară doar obținerea unei durate de execuție de un ordin de mărime inferior printr-o altă strategie de parcurgere a spațiului de căutare.

## Greedy.

În strategia backtracking căutarea soluției, adică vizitarea secvențială a nodurilor grafului soluțiilor cu revenire pe urma lăsată, se face oarecum "orbeşte" sau rigid, după o regulă simplă care să poată fi rapid aplicată în momentul "părăsirii" unui nod vizitat. În cazul metodei (strategiei) greedy apare suplimentar ideea de a efectua în acel moment o alegere. Dintre toate nodurile următoare posibile de a fi vizitate sau dintre toți paşii următori posibili, se alege acel nod sau pas care asigură un maximum de "cîştig", de unde şi numele metodei: *greedy = lacom*. Evident că în acest fel poate să scadă viteza de vizitare a nodurilor – adică a soluțiilor ipotetice sau a soluțiilor parțiale – prin adăugarea duratei de execuție a subalgoritmului de alegere a următorului nod după fiecare vizitare a unui nod. Există însă numeroşi algoritmi de tip greedy veritabili care nu conțin subalgoritmi de alegere. Asta nu înseamnă că au renunțat la alegerea greedy ci, datorită "scurtăturii" descoperite în timpul etapei de analiză a problemei, acei algoritmi efectuează la fiecare pas o alegere fără efort şi în mod optim a pasului (nodului) următor. Această alegere, **dedusă în etapa de analiză**, conduce la maximum de "profit" pentru fiecare pas şi scurtează la maximum drumul către soluția căutată.

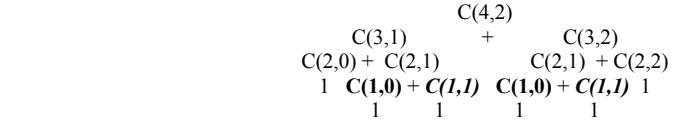
Aparent această metodă de căutare a soluției este cea mai eficientă, din moment ce la fiecare pas se trece dintr-un optim (parțial) într-altul. Totuşi, ea nu poate fi aplicată în general ci doar în cazul în care există certitudinea alegerii optime la fiecare pas, certitudine rezultată în urma etapei anterioare de analiză a problemei. Ori, dezavantajul este că, la majoritatea problemelor dificile, etapa de analiză nu poate oferi o astfel de "pistă optimă" către soluție. Un alt dezavantaj al acestei strategii este că nu poate să conducă către toate soluțiile posibile ci doar către soluția optimă (din punct de vedere a alegerii efectuate în timpul căutării soluției), dar poate oferi toate soluțiile optime echivalente.

## Programarea dinamică.

Este o metodă sau strategie ce îşi propune să elimine dezavantajele *metodei recursive* care, în ultimă instanță, am văzut că se reduce la parcurgerea în adîncime a arborelui apelurilor recursive (adică backtracking). Această metodă se apropie ca idee strategică de metoda Greedy, avînd însă unele particularități.

Pentru a o înțelege este necesară evidențierea dezavantajului major al recursivității. El constă din creşterea exagerată şi nerentabilă a efortului de execuție prin repetarea ineficientă a unor paşi. Urmărind arborele apelurilor recursive se observă repetarea inutilă a unor cazuri rezolvate anterior, calculate deja înainte pe altă ramură a arborelui. Metodă eminamente iterativă, programarea dinamică elimină acest dezavantaj prin "răsturnarea" procedeului de obținere a soluției şi implicit a arborelui apelurilor recursive. Printr-o abordare *bottom-up* (de la bază spre vîrf) ea reuşeşte să elimine operațiile repetate inutil în cazul abordării *top-down* (de la vîrf spre bază).

Cu toții am învățat că, dacă vrem să calculăm "cu mîna" o combinare sau un tabel al combinărilor, în loc să calculăm de fiecare dată combinări de $n$ luate cîte $k$ pe baza definiției recursive: *C(n,k)=C(n-1,k-1)+C(n-1,k) cînd n,k>0, sau, C(n,k)=1 cînd k=0 sau n=k*, este mult mai eficient să construim *Triunghiul lui Pascal*, pornind de la aceeaşi definiție a combinărilor.



1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
……..

Observaţi cum în arborele apelurilor recursive apar apeluri în mod repetat pentru calculul aceleaşi combinări. Acest efort repetat este evitat prin calcularea triunghiului lui Pascal în care fiecare combinare va fi calculată o singură dată.

În mod asemănător, aceeaşi diferenţă de abordare va exista între doi algoritmi de soluţionare a aceleaşi probleme, unul recursiv – backtracking - şi altul iterativ - proiectat prin metoda programării dinamice.

Dezavantajele acestei metode provin din faptul că, pentru a ţine minte paşii gata calculaţi şi a evita repetarea calculării lor (în termeni de grafuri, pentru a evita calcularea repetată a unor noduri pe ramuri diferite ale arborelui apelurilor recursive), este nevoie de punerea la dispoziţie a extra-spaţiului de memorare necesar şi de un efort suplimentar dat de gestiunea de memorie suplimentară.

## Branch & Bound.

Este strategia cea mai sofisticată de proiectare a algoritmilor. Ea a apărut datorită existenţei problemelor pentru care soluţia de tip backtracking poate necesita un timp astronomic de rulare a programului. În rezolvarea acestor probleme apare o asemenea penurie de informaţii încît modelul abstract asociat problemei - graful de căutare a soluţiilor – nu poate fi precizat în avans, din etapa de analiză. Singura soluţie care rămîne este includerea unui subalgoritm suplimentar ce permite construirea acestui graf pe parcurs, din aproape în aproape. Apariţia acelui subalgoritm suplimentar dă numele metodei: *branch&bound*.

Este posibilă compararea algoritmului branch&bound cu un robot ce învaţă să se deplaseze singur şi eficient printr-un labirint. Acel robot va fi obligat să-şi construiască în paralel cu căutarea ieşirii o hartă (un graf !) a labirintului pe care va aplica apoi , pas cu pas, metode eficiente de obţinere a drumului cel mai scurt.

La strategia de căutare a soluţiei în spaţiul (graful) de căutare - backtracking, fiecare pas urma automat unul după altul pe baza unei reguli încorporate, în timp ce la strategia greedy alegerea pasului următor era făcută pe baza celei mai bune alegeri. În cazul acestei strategii – branch&bound, pentru pasul următor algoritmul nu mai este capabil să facă vreo alegere pentru că este obligat mai întîi să-şi determine singur nodurile vecine ce pot fi vizitate. Numele metodei, *branch=ramifică* şi *bound=delimitează*, provine de la cele două acţiuni ce ţin locul acţiunii de alegere de la strategia Greedy. Prima acţiune este construirea sau determinarea prin ramificare a drumurilor de continuare, iar a doua este eliminarea continuărilor (ramurilor) ineficiente sau eronate. Prin eliminarea unor ramuri, porţiuni întregi ale spaţiului de căutare a soluţiei rămînînd astfel dintr-o dată delimitate şi “izolate”. Această strategie de delimitare din mers a anumitor “regiuni” ale spaţiului de căutare a soluţiilor este cea care permite reducerea ordinului de mărime a acestui spaţiu. Soluţia aceasta este eficientă doar dacă cîştigul oferit prin reducerea spaţiului de căutare (scăzînd *efortul suplimentar* depus pentru determinarea şi eliminarea din mers a continuărilor ineficiente) este substanţial.

Soluţiile de tip backtracking, avînd la bază un schelet atît de general (algoritmul de traversare a grafului de căutare a soluţiilor) sînt relativ simplu de adaptat în rezolvarea unor probleme. Poate acesta este motivul care a condus pe unii programatori lipsiţi de experienţă la convingerea falsă că *“Orice este posibil de rezolvat prin backtracking”*.

La ora actuală, lista problemelor pentru care nu se cunosc decît soluţii exponenţiale, total nerezonabile ca durată de execuţie a programului de soluţionare, cuprinde cîteva sute de probleme, una mai celebră ca cealaltă. Reamintim doar de “banala” (dar agasanta) Problemă a Orarului unei instituţii de învăţămînt care nu admite o soluţie backtracking datorită duratei astronomice de aşteptare a soluţiei.

Datorită totalei lor ineficienţe în execuţie, soluţiile backtracking obţinute după o analiză şi o proiectare “la prima mînă” (*brute-force approach,* în limba engleză) ajung să fie reanalizate din nou cu mai multă atenţie. Se constată atunci că modelul abstract asociat problemei, fie este prea sărac în informaţii pentru determinarea grafului de căutare a soluţiilor, fie conduce la un graf de căutare avînd dimensiunea nerezonabilă (exponenţială sau factorială, faţă de dimensiunea *n* a vectorului de intrare). Singura soluţie care rămîne în această situaţie la dispoziţie este ca aceste soluţii să fie reproiectate prin metoda branch&bound.

Un exemplu uşor de înţeles de "problemă branch&bound" îl oferă Problema Generală a Labirintului. Spre deosebire de Problema Labirintului prezentată anterior (care admitea o soluţie de tip backtracking), în varianta extinsă a acestei probleme, numărul direcţiilor posibile de urmat la fiecare pas poate fi oricît de mare, iar obstacolele pot avea orice formă şi dimensiune. În acest caz, singura posibilitate este construirea *"din mers"* a spaţiului de căutare a soluţiei. Astfel, pentru determinarea unui drum de ieşire din labirint sau a drumului cel mai scurt (dacă este posibilă determinarea acestuia în timp rezonabil!) este obligatorie adoptarea strategiei branch&bound.

Oferim în continuare o situaţie concretă, ilustrată. Sesizaţi că obstacolele, avînd forme şi dimensiuni diferite, nu pot fi ocolite decît pe un traseu "razant" sau pe un traseu ce urmează contorul exterior al acestora. Acest fapt complică mult problema şi impune luarea unor decizii "la faţa locului", în momentul întîlnirii şi ocolirii fiecărui obstacol, ceea ce *impune* o strategie de rezolvare de tip branch&bound – ramifică şi delimitează:

Deşi această strategie poate să crească uneori surprinzător de mult eficienţa algoritmilor de soluţionare (din nerezonabili ca timp de execuţie ei pot ajunge rezonabili, datorită reducerii dimensiunii exponenţiale a spaţiului de căutare a soluţiei), aplicarea ei este posibilă doar printr-un efort suplimentar în etapa de analiză şi în cea de proiectare a algoritmului. Dezavantajul major al acestei metode constă deci în efortul major depus în etapa de analiză a problemei (analiză care însă se va face o singură dată şi bine!) şi efortul suplimentar depus în etapa proiectării algoritmului de soluţionare.

Din experienţa practică este cunoscut faptul că, pentru a analiza o problemă dificilă un analist poate avea nevoie de săptămîni sau chiar luni de zile de analiză, în timp ce algoritmul de soluţionare proiectat va dura, ca timp de execuţie, doar cîteva zeci de minute. Dacă programul obţinut nu este necesar a fi rulat decît o dată, aceasta este prea puţin pentru "a se amortiza" costul mare al analizei şi proiectării sale. În acea situaţie, soluţia branch&bound este nerentabilă şi, probabil că ar fi mai ieftină strategia backtracking de soluţionare, chiar şi cu riscul de a obţine o execuţie (singura de altfel) a programului cu durata de o săptămînă (ceea ce poate să însemne totuşi economie de timp).

## Recursivitatea

Aşa cum am amintit deja, această metodă de programare poate fi privită ca formă particulară de exprimare a metodei Back-Tracking. Cu toate acestea, cei ce cunosc istoria informaticii şi originile programării ştiu că această metodă are totuşi un caracter special. Aceste lucruri dorim să le evidenţiem în continuare.

Încă înainte de apariţia primului calculator şi, deci implicit a oricărei tehnici de programare, unii matematicieni erau profund preocupaţi de noţiunea de calculabilitate. Această noţiune îi putea ajuta în efortul lor deosebit de a fundamenta noţiunea elementară de algoritm sau metodă automată de calcul. În paralel, cele mai valoroase rezultate le-au obţinut latino-americanul Alonso Church şi englezul Alan Turing. În timp ce Turing a introdus pentru algoritm modelul matematic abstract cunoscut sub numele de Maşina Turing (care stă la bazele modelului actual de calculator), Church a fundamentat noţiunea de metodă de calcul sau calculabilitatea pe funcţiile recursive. Astfel, teza lui Church afirma că *orice funcţie definită pe domeniul numerelor naturale este calculabilă dacă şi numai dacă ea este recursivă.* Deşi aparatul teoretic folosit de Church era în întregime matematic (se baza numai pe funcţii numerice naturale), lui nu i-a fost greu să demonstreze că orice algoritm nenumeric se reduce la funcţii recursive şi la mulţimi recursive de numere naturale (pe baza unor codificări riguros alese).

Acest din urmă rezultat este cel care ne interesează pe noi şi noi îl vom reformula fără ai afecta valabilitatea: orice algoritm poate fi rescris printr-un algoritm recursiv (limbajul de programare Lisp se bazează în întregime pe acest fapt). Chiar dacă nu constituie o demonstraţie riguroasă, următoarea echivalenţă practică (descrisă în pseudo-cod) este deosebit de convingătoare: orice instrucţiune de ciclare este echivalentă cu un apel recursiv de subprogram sau funcţie.

| Varianta iterativă-cu ciclu | Varianta cu apel recursiv |
|---|---|
| contor:=val_init;<br>Repetă<br>  Corp_ciclu;<br>  Incrementează(contor);<br>Pînă cînd contor=val_finală; | Funcție_Recursivă(contor){<br>  Dacă contor<val_finală atunci<br>  Corp_ciclu;<br>  Funcție_Recursivă(contor+1);<br>}<br>……<br>Funcție_Recursivă(val_init); // apelul inițial al funcției |

Observăm că în cazul variantei recursive condiția de scurt-circuitare a recursivității este echivalenta condiției de scurt-circuitare a ciclului. Gestiunea contorului se face în acest caz în background, prin mecanismul de stivă sistem.

Putem astfel concluziona: toți algoritmii iterativi pot fi înlocuiți prin algoritmi recursivi. Avantajul celor recursivi este dat de scăderea dimensiunilor programelor şi de creşterea lizibilității. Avantajul celor iterativi este viteza mărită de execuție prin gestionarea locală a parametrilor de ciclare (eliminînd-se astfel toate instrucțiunile push şi pop pentru gestionarea stivei).

Spre edificare, vă oferim în continuare cîteva probleme clasice (simple) rezolvate în C prin metoda recursivă. În capitolul cu probleme ce necesită back-tracking veți găsi şi alte soluții recursive (în C) ale unor probleme ceva mai dificile; astfel se vor putea sesiza mai bine avantajele acestei metode "naturale" de programare. (Întrucît am considerat acest capitol ca fiind destul de "tehnic", prezentăm în continuare doar variantele de program în limbajul C, ce este considerat mai "tehnic" decît limbajul Pascal.)

1. Să se copieze un şir de caractere într-altul.

```
#include <stdio.h>

char *sir1="primul",*sir2="al doilea";

void strcopy(char *sursa,char *dest){
 if ((*dest++=*sursa++)==NULL) return;
 else strcopy(sursa,dest);
}
void main(void){
 printf("\nInainte, sirul sursa:%s, sirul destinatie:%s",sir1,sir2);
 strcopy(sir1,sir2);
 printf("\nSi dupa, sirul sursa:%s, sirul destinatie:%s",sir1,sir2);
}
```

2. Să se afişeze primele n pătrate perfecte.

```
#include <stdio.h>
#include <math.h>

int n;

void patrat(int m){
 if(m>n)return;
 else {
   printf("%i:%i ",m,m*m);
   patrat(m+1);
 }
}
void main(void){
 printf("n=");scanf("%i",&n);
 patrat(1);
}
```

3.Algoritmul lui Euclid.

```
#include <stdio.h>

int cmmdc(int m,int n){
 if (n==0) return(m);
 else cmmdc(n,m%n);
}

void main(void){
 int m,n;
 printf("m,n=");scanf("%i,%i",&m,&n);
 printf("cmmdc(%i,%i)=%i",m,n,cmmdc(m,n));
}
```

4. Se citeşte n, să se găsească primul număr prim mai mare decît n. (Se presupune cunoscută demonstrația faptului că există p-prim mai mare decît oricare n. Sîntem astfel siguri că algoritmul se opreşte! )

```
#include <stdio.h>
#include <math.h>

int n;

int are_divizor(int p,int d){
 if(d>sqrt(p))return 0;
 else if(p%d==0) return 1;
   else are_divizor(p,d+1);
}
void prim(int p){
 if(!are_divizor(p,2)){
   printf("\n%i",p);
   return;
 }
 else prim(p+1);
}
void main(){
 printf("n=");scanf("%i",&n);
 prim(n+1);
}
```

5. Să se afişeze primele n numere prime.

```
#include <stdio.h>
#include <math.h>

int n;

int are_divizor(int p,int d){
 if(d>sqrt(p))return 0;
 else if(p%d==0) return 1;
   else are_divizor(p,d+1);
}
void prim(int p,int i){
 if(i>n)return;
 if(!are_divizor(p,2)){
   printf("%i,",p);
   prim(p+1,i+1);
```

```
 }
 else prim(p+1,i);
}
void main(){
 printf("n=");scanf("%i",&n);
 prim(2,1);
}
```

6. Se citeşte n gradul unui polinom P şi a[0],a[1],...,a[n] coeficienţii reali ai acestuia. Se citeşte o valoare reală x, să se calculeze P(x).

```
#include <stdio.h>

int n;
float a[20],x;

float P(int i){
 if(i==1)return a[0];
 else return P(i-1)*x+a[i-1];
}
void citeste_coef(int i){
  if(i>n)return;
  else {printf("%i:",i);scanf("%f",&a[i]);citeste_coef(i+1);}
}
void main(){
 printf("n=");scanf("%i",&n);
 citeste_coef(0);
 printf("x=");scanf("%f",&x);
 printf("P(%f)=%f",x,P(n+1));
}
```

7. Se citesc m şi n gradele a două polinoame P şi Q, şi a[0],a[1],...,a[m] respectiv b[0],b[1],...,b[n] coeficienţii reali ai acestora. Să se afişeze coeficienţii c[0],c[1],...,c[m+n] ai polinomului produs R=PxQ.

```
#include <stdio.h>

int m,n;
float a[20],b[20],c[40];

float suma_prod(int i,int j){
 if(j==i)return a[i]*b[0];
 else return a[i-j]*b[j]+suma_prod(i,j+1);
}
void calc_coef(int i){
 if(i>m+n)return;
 else c[i]=suma_prod(i,0);
}
void citeste_coef(float a[],int i){
  if(i>n)return;
  else {printf("%i:",i);scanf("%f",&a[i]);citeste_coef(a,i+1);}
}
void afis_coef(float a[],int i){
  if(i>n)return;
  else {printf("%f ",a[i]);afis_coef(a,i+1);}
}
void main(){
 printf("m(gradul polinomului P)=");scanf("%i",&m);
 printf("Introd.coef.polinomului P:");
 citeste_coef(a,0);
```

```
 printf("n(gradul polinomului Q)=");scanf("%i",&n);
 printf("Introd.coef.polinomului Q:");
 citeste_coef(b,0);
 calc_coef(0);
 afis_coef(c,0);
}
```

8. Se citeşte n, o valoarea întreagă pozitivă, să se determine suma cifrelor lui n.

```
#include <stdio.h>

int n;

int suma(int n){
 if(n<10)return n;
 else return n%10+suma(n/10);
}
void main(){
 printf("n=");scanf("%i",&n);
 printf("suma cifrelor=%i",suma(n));
}
```

## Probleme rezolvate şi exerciţii de programare

Vom începe prin a face o observaţie importantă: există totdeauna un pericol în oferirea "pe tavă" a soluţiilor la probleme. În următoarele subcapitolele nu am fost deloc "zgîrciţi" şi se pot găsi destule probleme rezolvate atît în Pascal cît şi în C, deşi pentru unele veţi putea găsi rezolvarea doar într-unul din limbaje. Pericolul constă în faptul că, începătorilor leneşi, lipsiţi de voinţă şi înclinaţi către a copia mereu, li se oferă posibilitatea să nu-şi mai "bată" capul cu rezolvarea problemelor acum cînd au totul "de-a gata". Desigur, cei care ies în pierdere sînt tot ei. Ne-am asumat acest risc gîndindu-ne nu atît la cei leneşi cît mai ales la programatorii începători bine-intenţionaţi cărora aceste probleme, cu rezolvările lor tipice, le poate fi de un real folos. Putem spune că urmat astfel exemplul autorilor mediilor de programare Turbo Pascal şi Turbo C (sau Borland) care prin help-urile lor generoase au contribuit decisiv la formarea multor generaţii de programatori.

Vă avertizăm că, în practica concretă de programare, programatorul (care nu este şi analist) primeşte de la cel care a analizat înainte problema doar *indicaţii de programare*. Rareori analistul pune la dispoziţia programatorului şi o descriere în pseudocod a algoritmilor ce trebuiesc implementaţi. Deci, nici un programator începător nu trebuie să-şi facă iluzia că "generozitatea" din acest capitol o va mai întîlni vreodată în practica concretă de programare sau că va avea vreodată la dispoziţie surse "abundente" de inspiraţie. Este cert că în practică lipsa "inspiraţiei" va trebui compensată prin "transpiraţie".

## Probleme elementare. Exerciţii de programare

Oferim în continuare o mulţime de probleme de programare "clasice" rezolvate într-un mod didactic. Am adăugat înaintea celor două versiuni de soluţionare în cele două limbaje de programare, Pascal şi C, cîteva rînduri ce cuprind elementele de bază ale analizei probleme.

Ne-am străduit să aşezăm problemele în ordinea dificultăţii lor, de la cele elementare spre cele mai dificile. De aceea este recomandat ca ele să fie parcurse în această ordine.

Atragem atenţia începătorilor: una din trăsăturile specifice ale programării este că o problemă admite mai multe rezolvări corecte. Deşi pot fi diferite în unele detalii, fiind echivalente prin rezultatele pe care le oferă, noi le vom numi *variante*. Aşa că, ceea ce se oferă în continuare este doar o variantă de rezolvare pentru fiecare problemă, ea fiind pasibilă de îmbunătăţiri, atît pentru versiunea Pascal cît şi pentru versiunea C. Se zice că o variantă de program (algoritm) este mai *eficientă* decît alta dacă cantitatea de *resurse-calculator* folosită este mai redusă: memorie-calculator (necesarul de spaţiu) mai puţină şi timp-calculator (necesarul de timp sau durata de execuţie) mai mic.

Este cunoscut că în învăţarea unei limbi străine ajută mult exersarea traducerilor dintr-o limbă într-alta. Evident, pentru realizarea retroversiunii (termenul de specialitate folosit) este necesară cunoaşterea temeinică a uneia din cele două limbaje. La fel, în cazul programării, învăţarea celui de-al doilea limbaj de programare este mult uşurată de faptul că am asimilat deja primul limbaj de programare. În finalul capitolului vor apare, pentru exerciţiu, mai multe probleme avînd varianta de rezolvare doar într-unul din limbaje, Pascal sau C, şi vă propunem să scrieţi programul corespondent în celălalt limbaj. Astfel, cei care au învăţat deja Pascal vor putea astfel să înveţe C-ul foarte rapid , şi reciproc.

**Să se afişeze soluţiile reale ale ecuaţiei de gradul al doilea.**

**Analiza problemei - elaborarea algoritmului:**

Fie ecuatia de gradul II $ax^2+bx+c=0$

-daca toti coeficientii ecuatiei sunt egali cu 0 atunci avem o ecuatie nedeterminata care are o infinitate de solutii (S=R).

-daca a,b=0 ,iar c<>0 atunci avem o ecuatie care nu are solutii.

-daca a=0 ,b,c <>0 atunci ecuatia se reduce la o ecuatie de gradul I care are o singura solutie x=-c/b.

-daca a,b,c <>0 atunci trebuie calculat discriminantul (delta) ecuatiei d=b*b-4*a*c

-daca d>=0 atunci ecuatia are solutii reale $x_{1,2}$=(-b+-sqrt(d))/(2*a)

-daca d<0 atunci ecuatia nu are solutii reale.

program ecuatie;

```
var a,b,c,d:real;
BEGIN
write('a=');readln(a);
write('b=');readln(b);
write('c=');readln(c);
if a=0 then
  if b=0 then
    if c=0 then
      writeln('Ecuatie nedeterminata, S=R')
    else writeln('Ecuatia nu are solutii.')
  else writeln('Ecuatie de gradul I cu solutia x=',-c/b:6:2)
else
  begin
    d:=b*b-4*a*c;
    if  d>=0 then
       begin
         writeln('x1=',(-b-sqrt(d))/(2*a):6:2);
         writeln('x2=',(-b+sqrt(d))/(2*a):6:2);
       end
    else writeln('Ecuatia nu are solutii reale.');
  end;
readln;
END.
```

```
#include <stdio.h>
#include <math.h>

float a,b,c;  // coeficientii ecuatiei de gradul II
float delta;

void main(){
printf("Introd.coefic.ecuatiei a b c:");scanf("%f %f %f",&a,&b,&c);
delta=b*b-4*a*c;
if (delta>=0) {
 printf("Sol.reale: x1=%6.2f, x2=%6.2f",(-b+sqrt(delta))/2./a,(-b-sqrt(delta))/2./a);
} else {
 printf("Sol.complexe: x1=(%6.2f,%6.2f), x2=(%6.2f,%6.2f)",-b/2./a,sqrt(-delta)/2./a,-b/2/a,-
1./2./a*sqrt(-delta));
}
}
```

**Să se determine dacă trei numere reale pot reprezenta laturile unui triunghi. Dacă da, să se calculeze perimetrul si aria sa.**

**Analiza problemei – elaborarea algoritmului :**

-trebuie sa vedem cînd trei numere pot fi lungimile laturilor unui triunghi: cele trei numere trebuie sa fie pozitive si suma a oricare doua dintre ele sa fie mai mare decat a treia latura.
-algoritmul poate fi implementat folosind o functie care sa verifice daca cele trei numere indeplinesc conditiile enumerate mai sus.
-dupa verificarea celor trei numere calculam perimetrul si aria triunghiului folosind formula lui Heron s=sqrt(p(p-a)(p-b)(p-c)), unde semiperimetrul este p=(a+b+c)/2.

```
program arie;
var a,b,c:integer;
  s,p:real;

function laturi_ok:boolean;
begin
   laturi_ok:= (a>0) and (b>0) and (c>0) and  (a+b>c) and (a+c>b) and (b+c>a) ;
```

```
end;

BEGIN
write('introduceti laturile');readln(a,b,c);
P:=(a+b+c)/2;
IF laturi_ok then
begin s:=sqrt(p*(p-a)*(p-b)*(p-c));
    writeln('s=',s:5:2);
    writeln('p=',p*2:5:2);
end
else writeln('laturi negative sau prea mari');
readln;
END.

// solutia in limbajul C

#include <stdio.h>
#include <math.h>

float a,b,c;
float s,p;

int laturi_ok(void){
   return (a>0)&&(b>0)&&(c>0)&&(a+b>c)&&(a+c>b)&&(b+c>a) ;
}

void main(void){
printf("introduceti laturile a b c:");scanf("%f %f %f",&a,&b,&c);
p=(a+b+c)/2;
if (laturi_ok()){
    s=pow(p*(p-a)*(p-b)*(p-c), 0.5);
    printf("s=%6.2f p=%6.2f",s,p);
}
else printf("laturi negative sau prea mari");
}
```

**Să se afişeze media aritmetică, geometrică şi hiperbolică a trei valori reale.**

**Analiza problemei - elaborarea algoritmului:**

-trebuie aplicate formulele pentru calculul celor trei medii si trebuie analizate cazurile :

- cand nu putem calcula media geometrica a trei numere(cand produsul lor este negativ,deci cand avem unul sau trei numere negative)
- cand nu putem calcula media hiberbolica a numerelor(cand unul dintre numere este egal cu 0 si nu poate fi facuta impartirea cu 0).

- in TurboPascal exista o functie pentru calculul radicalului de ordinul 2 (sqrt),dar pentru calculul radicalului de ordinul n nu este implementata o functie de aceea pentru calculul radicalului de ordinul n folosim functia exponentiala ( exp ) pentru a calcula o puterea a lui: $a^n = \exp(n*\ln(a))$, iar pentru a calcula radical de ordinul n din a: $a^{1/n}=\exp(1/n*\ln(a))$ .

```
program medii;
var a,b,c,ma,mg,mh:real;

BEGIN
write('a=');readln(a);
write('b=');readln(b);
write('c=');readln(c);
writeln('ma=',(a+b+c)/3:6:2);
if (a=0) or (b=0) or (c=0)  then writeln('mg=0')
```

```
else
 if a*b*c>0 then writeln('mg=',exp(1/3*ln(a*b*c)):6:2)
   else writeln('Nu putem calcula media geometrica ,nr negative .');
if (a=0) or (b=0) or (c=0) then writeln('Nu putem calcula media hiperbolica')
else writeln('mh=',3/(1/a+1/b+1/c):6:2);
readln;
END.

// solutia in limbajul C

#include <stdio.h>
#include <math.h>

float a,b,c,ma,mg,mh;

void main(void){
printf("a=");scanf("%f",&a);
printf("b=");scanf("%f",&b);
printf("c=");scanf("%f",&c);
printf("ma=%6.3f",(a+b+c)/3.);
if (a==0 || b==0 || c==0){
            printf("mg=0");
            printf("Nu putem calcula media hiperbolica");
} else {
   if (a*b*c>0) then writeln("mg=%6.3f",pow(a*b*c,1./3.));
   else printf("Nu putem calcula media geometrica ,nr negative .");
   printf("mh=%6.3f",3./(1/a+1/b+1/c));
       }
}
```

**Să se determine suma primelor n numere naturale.**

**Analiza problemei – elaborarea algoritmului:**

-suma primelor n numere naturale poate fi calculata, fără a folosi formula cunoscută, cu una dintre instructiunile repetitive cunoscute(for,while ,repeat)

-indiferent de instructiunea repetitiva folosita trebuie initializata suma cu 0 (s=0)

-folosim un contor i (1,n) care la fiecare pas se incrementeaza cu 1 si se aduna la s

-ciclul se incheie cand valoarea lui i>n

-daca folosim instructiunea for, numarul pasilor este cunoscut, valoarea initiala a contorului fiind 1, iar cea finala fiind n.

```
program suma;
var  s,i:word;

BEGIN
writeln('Introduceti limita n=');readln(n);
s:=0;
for i:=1 to n do  s:=s+i;
writeln('s=',s);
readln;
END.

// solutia in limbajul C

#include <stdio.h>

unsigned s,i;
void main(void){
printf("Introduceti n=");scanf("%u",&n);
```

```
for(s=0,i=1;i<=n;i++) s+=i;
printf("s=%u",s);
}
```

**Să se determine dacă n este pătrat sau cub perfect.**

**Analiza problemei – elaborarea algoritmului:**

-pentru a verifica daca un numar este patrat perfect calculam radacina patrata a numarului
-daca numarul este patrat perfect radacina lui este un numar intreg altfel este un numar cu zecimale
-verificam daca patratul partii intregii a radacinii numarului este egal cu numarul dat ,daca da numarul este patrat perfect altfel numarul nu este patrat perfect
-la fel procedam pentru a verifica daca un numar este cub perfect .

```
program patrat_si_cub_perfect;
var n:longint;

BEGIN
write('n=');readln(n);

if n=trunc(sqrt(n))*trunc(sqrt(n))  then
   writeln(n,'  este patrat perfect')
else
   writeln(n,'  nu este patrat perfect');

if n=trunc(exp(1/3*ln(n)))*trunc(exp(1/3*ln(n)))*trunc(exp(1/3*ln(n))) then
  writeln(n,'  este cub perfect')
else
  writeln(n,'  nu este cub perfect');
readln;
END.

// solutia in limbajul C

#include <stdio.h>
#include <math.h>

unsigned long n,m;

void main(void){
printf("n=");scanf("%lu",&n);
m=pow(n,0.5);if(n==m*m) printf("n este patrat perfect.");
m=pow(n,1./3.);if(n==m*m*m) printf("n este cub perfect.");
}
```

**Să se determine toate numerele de 4 cifre  divizibile cu n .**

**Analiza problemei - elaborarea algoritmului:**

-observam ca daca abordam solutia la "prima mînă" numarul paşilor în cadrul ciclului for este de 8999, pentru ca valoarea de intrare in ciclul for este 1000  iar  valoarea de iesire este 9999.
-re-analizînd problema putem stabili un numar foarte mic de pasi care este egal cu numarul de numere formate din patru cifre divizibile cu n .

```
program nr_divizibile;
var n,i:word;

BEGIN
```

```
write('n=');readln(n);
{for i:=1000 to 9999 do
  if (i mod n=0)  then write(i,'  ');
writeln;}

if 1000 mod n =0 then
 for i:=(1000 div n) to 9999 div n do
  write(i*n,',')
else
 for i:=(1000 div n)+1 to 9999 div n do
  write(i*n,',');
readln;
END.


// solutia in limbajul C

#include <stdio.h>

unsigned n,i;

void main(void){
printf("n=");scanf("%u",&n);
if (1000 % n ==0)
 for(i=1000 /n;i<=9999 / n;i++) pritnf("%4u,",i*n);
else
 for(i=1000 / n+1;i<= 9999 / n;i++) printf("4u,",i*n);
}
```

**Să se determine suma cifrelor lui n.**

**Analiza problemei - elaborarea algoritmului:**

-suma cifrelor numarului citit se obtine adunînd de fiecare data ultima cifra ce este restul impartirii lui n la 10 (n mod 10) iar ceea ce ramine eliminind ultima cifra este dat de impartirea lui n la 10 (n div 10).

```
program suma_cifre;
var n,s:word;

BEGIN
write('n=');readln(n);
s:=0;
while n<> 0 do
begin
  s:=s+n mod 10;
  n:=n div 10;
end;
writeln('s=',s);
readln;
END.


// solutia in limbajul C

#include <stdio.h>

unsigned n,s;

void main(void){
```

```
printf("n=");scanf("%u",&n);
s=0;
while (n!=0) {
  s+=n % 10;
  n/=10;
}
printf("s=%u",s);
}
```

**Să se afişeze următorul triunghi de numere:**

**1**
**1 2**
**1 2 3**
**.....**
**1 2 3 ..n**

```
program triunghi;
var i,j,n:word;

BEGIN
write('n=');readln(n);
for i:=1 to n do
  begin
  for j:=1 to i do
    write(j,' ');
  writeln;
  end;
readln;
END.
```

```
// solutia in limbajul C

#include <stdio.h>

int n,i,j;

void main(void){
printf("n=");scanf("%u",&n);
for(i=1;i<=n;i++) {
  for(j=1;j<=i;j++)
    printf("%i ",j);
  putchar('\n');
  }
}
```

**Se citeşte o valoare reală. Să se determine radical din *x* cu 5 zecimale exacte pe baza şirului convergent $x_n=1/2(x_n-1+x/x_{n-1})$, cu $x_0>0$ arbitrar ales.**

**Analiza problemei – elaborarea algoritmului:**

Pentru rezolvarea problemei folosim sirul convergent dat (metoda lui Newton) care consta in etapele:
-pornind cu $x_0=1$ se genereaza recursiv urmatorul sir de numere reale

$$x_n=1/2(x_n-1+x/x_{n-1})$$

-cand diferenta intre $x_n$ si $x_{n-1}$ este foarte mica(mai mica decat o limita data)procesul de generare a lui $x_n$ inceteaza
-la sfarsit $x_n$ reprezinta radacina patrata a lui x.

```
var x,xn,xn_1:real;
```

```
BEGIN
write('Introduceti valoarea:');readln(x);
xn:=1;
repeat
  xn_1:=xn;
 xn:=0.5*(xn_1+x/xn_1);
until abs(xn-xn_1)<1e-5;
writeln('radical din ',xn:6:2,'=',sqrt(x):10:5);
readln;
END.

// solutia in limbajul C

#include <stdio.h>
#include <math.h>

float x,xn,xn_1;

void main(void){
printf("Introduceti valoarea:");scanf("%f",&x);
xn=1;
do{
  xn_1=xn;
 xn=0.5*(xn_1+x/xn_1);
} while abs(xn-xn_1)<1e-5;
printf('radical obtinut =%7.5f, comparativ cu %7.5",x,pow(x,0.5));
}
```

**Se citeşte n, să se determine toate numerele perfecte mai mici decît n. Un număr este perfect dacă este egal cu suma divizorilor săi (de ex. 6=1+2+3).**

**Analiza problemei – elaborarea algoritmului:**

-pentru a verifica daca un numar este patrat perfect trebuie sa –i determinam divizorii si sa verificam daca suma acestora este egala cu n
- se observa ca ultimul divizor nu trebuie luat in calcul pentru ca este egal cu n
-pentru a afisa toate numerele perfecte < n folosim un ciclu while in care il decrementam pe n si verificam daca noul n este un numar perfect ,daca da il afisam

```
program nr_perfecte;
var n,d,i:word;

BEGIN
write('n=');readln(n);
while n>1 do
 begin
   dec(n);
   d:=0;
   for  i:=1 to n-1 do
    if n mod i=0 then  d:=d+i;
    if n=d then writeln(n);
 end;
readln;
END.
// o varianta C

#include <conio.h>
#include <stdio.h>
```

```
main()
{
 long int i,n,j,sum,k;
 clrscr();
 printf("n=");
 scanf("%ld",&n);
 k=0;
 i=0;
 do
 {
  k=k+1;
  do
  {
   sum=1;
   i=i+1;
   for(j=2;j<=i/2;j++)
    if (i%j==0)
     sum=sum+j;
  }
  while(sum!=i);
  printf("%ld ",i);
 }
 while(k<n);
}
```

**Se citeşte *n* un număr întreg pozitiv, să se afişeze *n* transcris în baza 2.**

**Analiza problemei - elaborarea algoritmului:**

- folosim algoritmul cunoscut :
  cît timp n <>0 executa
  - imparte n la 2
  - in urma impartirii n retine catul si restul
- numarul in baza doi se obtine scriind resturile in ordinea inversa in care le-am obtinut
- pentru a retine aceste resturi care trebuie tiparite in ordine inversa am folosit un sir (n2inv) in care am retinut resturile pe care dupa aceea l-am afisat in ordine inversa.

```
program transf_in_baza_2;
var n,n2,i,j:word;
  n2inv:array[1..20] of word;

BEGIN
write('n=');readln(n);
i:=1;
while n>0 do
begin
 n2:=n mod 2;
 n2inv[i]:=n2;
 n:=n div 2;
 i:=i+1;
end;
for j:=i-1 downto 1 do
  write(n2inv[j]);
readln;
END.
```

```
// o varianta C putin diferita
```

```
#include <stdio.h>
typedef unsigned char pointer[4];
void afiseaza(pointer px,int dim,char* format){
 int i,j;
 for(j=dim-1;j>=0;j--){
   for(i=8;i>=0;i--) printf("%c",px[j] & (1<<i) ? '1':'0');
   putchar('.');
   }
 printf(" adica ");printf(format,*px);
}
 float y;
 long x;

void main(void){
 printf("\nIntrod. intregul x si realul y:");scanf("%d %f",&x,&y);
 printf("\nx= ");
 afiseaza(&x,sizeof(x),"%d");
 printf("\ny= ");
 afiseaza(&y,sizeof(y),"%f");
}
```

**Se citeşte *n* şi şirul de valori reale *x1,x2,..,xn* ordonate crescător. Să se determine distanța maximă între două elemente consecutive din şir.**

**Analiza problemei - elaborarea algoritmului :**

- este o problema maxim
- distanta dintre primele valori consecutive din sir se noteaza cu max
- dupa care facem o comparatie cu urmatoarele distante dintre valori
- in momentul in care se intalneste o valoare mai mare decat max atunci aceasta valoare va deveni noul max
- algoritmul se opreste in momentul in care se face comparatia dintre max si distanta dintre ultimele doua valori ale sirului.

```
program dist_elem;
var n,i:word;
    max:real;
    x:array[1..50] of real;

BEGIN
write('n=');readln(n);
for i:=1 to n do
 begin
   write('x[',i,']=');
   readln(x[i]);
 end;
max:=x[2]-x[1];
for i:=2 to n-1 do
  if x[i+1]-x[i]>max then max:=x[i+1]-x[i];
writeln('max=',max:6:2);
readln;
END.
```

**Se citeşte *n* gradul unui polinom şi şirul coeficienților $a_n, .. , a_0$. Se citeşte *x*, să se determine P(x).**

```
program polinom;
var n,i :integer;
    p,x:real;
    a:array[0..20] of integer;
```

```
BEGIN
write('n=');readln(n);
for i:=0 to n do
begin
  write('a[',i,']=');
  readln(a[i]);
end;
write('x=');readln(x);

{p:=a[0];
for i:=1 to n do
  p:=p+a[i]*exp(i*ln(x));
writeln('P(',x,')=',p:6:2);}

p:=0;
for i:=n downto 0 do
  p:=p*x+a[i];
writeln('P(',x,')=',p:6:2);
readln;
END.
```

**Se citeşte o propoziție (şir de caractere) terminată cu punct. Să se determine cîte vocale şi consoane conține propoziția.**

**Analiza programului - elaborarea algoritmului:**

- citim propozitia caracter cu caracter pana la intalnirea caracterului '.'
- folosim instructiunea case (selectie multipla) care daca la intalnirea unei vocale din sir incrementeaza nr de vocale ,iar la intalnirea unei consoane incrementeaza nr de consoane.

```
program nr_consoane_si_vocale;
var c:char;
    i,nv,nc:word;
    sir:string[25];

BEGIN
write('Introduceti propozitia:');readln(sir);
i:=1; nv:=0; nc:=0;
repeat
 case sir[i] of
   'a','e','i','o','u': nv:=nv+1;
   'b','c','d','f','g','h','j','k','l','m','n','p','r','s','t','x','y','w' :
     nc:=nc+1;
   end;
   i:=i+1;
until sir[i]='.';
writeln('Nr de vocale=',nv);
writeln('Nr de consoane=',nc);
readln;
END.

// varianta C

#include <stdio.h>
#include <ctype.h>

int i,vocale=0,consoane=0;
char c,sir[80];

void main(void){
```

```
  printf("Introd.propozitia terminata cu punct:");gets(sir);
  for(i=0;sir[i]!='.';i++)
    switch (toupper(sir[i])){
      case 'A':
               case 'E':
               case 'I':
               case 'O':
               case 'U': vocale++; break;
      default: if (isalpha(sir[i])) consoane++;
    }
  printf("Vocale:%i, Consoane:%i, Alte car.:%i", vocale, consoane, i-vocale-consoane);
}
```

**Se citeşte *m,n* dimensiunea unei matrici $A=(a[i,j])_{mxn}$ de valori reale. Să se determine suma elementelor pe fiecare linie şi coloană.**

```
program matrice3;
var m,n,i,j:word;
   a:array[1..50,1..50] of real;
   sl,sc:array[1..50] of real;

BEGIN
write('Introduceti nr de linii m=');readln(m);
write('Introduceti nr de coloane n=');readln(n);
for i:=1 to m do
  begin
   for j:=1 to n do
     begin
       write('a[',i,',',j,']=');
       read(a[i,j]);
     end;
   writeln;
  end;

for i:=1 to m do sl[i]:=0;
for j:=1 to n do sc[j]:=0;

for i:=1 to m do
  begin
    for j:=1 to n do
      sl[i]:=sl[i]+a[i,j];
    writeln('suma elementelor de pe linia ',i,'=',sl[i]:6:2);
  end;

for j:=1 to n do
  begin
    for i:=1 to m do
      sc[j]:=sc[j]+a[i,j];
    writeln('suma elementelor de pe coloana ',j,'=',sc[j]:6:2);
  end;
readln;
END.

// varianta C

#include <stdio.h>

unsigned m,n,i,j;
float a[50][50];
```

```
float sl[50],sc[50];

void main(void){
printf("Introduceti nr de linii m=");scanf("%u",&m);
pritnf("Introduceti nr de coloane n=");scanf("%u",&n);
for (i=0;i<m;i++){
  for (j=0;j<n;j++){
     printf("a[%u,%u]=",i,j);
     scanf("%f",&a[i][j]);
  }
  putchar('\n');
};
for (i=0;i<m;i++) sl[i]=0;
for (j=0;j<n;j++) sc[j]=0;
for (i=0;i<m;i++) {
   for (j=0;j<n;j++)
     sl[i]+=a[i][j];
   printf("suma elementelor de pe linia %u =%f",i,sl[i]);
}
for (j=0;j<n;j++) {
   for (i=0;i<m;i++)
     sc[j]+=a[i][j];
   pritnf("suma elementelor de pe coloana %u=%f",j,sc[j]);
}
}
```

**Se citeşte *n* şi *k,* şi o matrice $A=a[i,j]_{nxn}$ pătratică. Să se determine $A^k$.**

**Analiza problemei - elaborarea algoritmului:**

-algoritmul consta de fapt in calcularea elementelor matricii produs

-elementul c[i,j] =suma(k=1..n) a[i,k]*b[i,k] .

-$A^k$=A*A*..*A

-matricea fiind patratica atunci cand k=2 termenii b[i,k]=a[i,k],iar cand k>2 termenii b[i,k] pastreaza elementele produsului anterior A*A, folosim pentru aceasta atribuire procedura aribuire.

```
program matrice1;
type matrice= array[1..3,1..3] of real;
var a,b,p: matrice;
   n,i,j,k,l:word;

procedure atribuire(a:matrice);
begin
 for i:=1 to n do
   for j:=1 to n do
     b[i,j]:=a[i,j];
end;

procedure inmultire ;
begin
 for i:=1 to n  do
  for j:=1 to n do
    p[i,j]:=0;
 for i:=1 to n do
     for j:=1 to n do
       for l:=1 to n do
         p[i,j]:=p[i,j]+a[i,l]*b[l,j];
end;

BEGIN
```

```
write('Introduceti puterea lui A ,k=');readln(k);
write('Introduceti dimensiunea matricii n=');readln(n);
for i:=1 to n do
  begin
    for j:=1 to n do
      begin
        write('a[',i,',',j,']=');
        readln(a[i,j]);
      end;
    writeln;
  end;

if k=1 then
  for i:=1 to n do
      for j:=1 to n do
          p[i,j]:=a[i,j]
else
 if k=2 then
  begin
    atribuire(a);
    inmultire;
  end
else
  begin
    atribuire(a);
    inmultire;
    k:=k-1;
    while  k>1 do
      begin
        atribuire(p);
        inmultire;
        k:=k-1;
      end;
  end ;


 for i:=1 to n do
  begin
    for j:=1 to n do
       write('p[',i,',',j,']=',p[i,j]:6:2,'  ');
    readln;
  end;
readln;
END.
```

**Iată un program Pascal care gestionează cu ajutorul unui fişier un catalog de note şi persoane.**

```
Type Persoana=Record Nume:String[20];Nota:Array[1..4]of integer; End;
Var f:File of Persoana;
    Perstemp:Persoana;
Procedure Creare;
Begin
    Writeln('Introd.');
    Assign(f,'Test.jo');
    Rewrite(f);
    Repeat
        With PersTemp do begin
            Write('Numele:');Readln(Nume);
            If Nume='' then break;
```

```
        Write('Notele:');Readln(Nota[1],Nota[2],Nota[3],Nota[4]);
      end;
      Write(f,PersTemp);
   Until False;
   Close(f);
End;
Procedure Citire;
Begin
   Writeln('Introd.');
   Assign(f,'Test.jo');
   Reset(f);
   Repeat
      Read(f,PersTemp);
      With PersTemp do begin
         Writeln('Numele:',Nume);
         Writeln('Notele:',Nota[1],Nota[2],Nota[3],Nota[4]);
      end;
   Until Eof(f);
   Close(f);
End;
BEGIN
  Creare;
  Citire;
END.
```

**Iată trei programe care exemplifică modul de lucru cu fişiere în limbajul C.**

**// Copierea unui fisier text sursa intr-un fisier destinatie**

```
#include <stdio.h>
void main(void)
{
  FILE *in, *out;
  char numfin[20],numfout[20];
  long contor=0;

  printf("Nume  fisier  sursa:");gets(numfin);
  printf("Nume fis.destinatie:");gets(numfout);
  if ((in = fopen(numfin, "rt"))== NULL){
    fprintf(stderr, "Eroare: %s fisier inexistent.\n",numfin);
    return;
  }
  out = fopen(numfout, "wt");
  while (!feof(in)){
    fputc(fgetc(in), out);contor++;
  }
  fclose(in);fclose(out);
  printf("Lungimea fis.destinatie este de %ld octeti.",contor);
}
```

**// Copierea unui fisier text sursa intr-un fisier destinatie**
**// cu substituirea unor cuvinte date prin linia de comanda**

```
#include <stdio.h>
void main(int argc,char *argv[])
{
  FILE *in, *out;
  char numfin[20],numfout[20],c;
```

```
  unsigned i=0,contor=0;

  printf("Nume  fisier  sursa:");gets(numfin);
  printf("Nume fis.destinatie:");gets(numfout);
  if ((in = fopen(numfin, "rt"))== NULL){
    fprintf(stderr, "Eroare: %s fisier inexistent.\n",numfin);
    return;
  }
  out = fopen(numfout, "wt");
  while (!feof(in)){
    if((c=fgetc(in))==argv[1][i]){
          if(argv[1][++i]==0)  // s-a detectat sfirsitul sirului de caractere
            fputs(argv[2],out),i=0;  // se scrie sirul de caractere inlocuitor
    }
    else fputc(c, out);contor++;
  }
  fclose(in);fclose(out);
  printf("Lungimea fis.destinatie este de %d octeti.",contor);
}
```

**// prelucrarea unul fisier C ce contine o agenda telefonica**

```
#include <stdio.h>
#include <ctype.h>
#include <conio.h>

struct articol
{
 char nume[10],adresa[10],tel[7];
} inreg;

FILE *fagenda,*ftemp;
char mod[3]="wb";

void creare(void){
 char temp;
 puts("\nCrearea agendei:");
 do{
  printf("\nNume:");gets(inreg.nume);
  printf("Adresa:");gets(inreg.adresa);
  printf("Tel:");gets(inreg.tel);
  fwrite(&inreg, sizeof(inreg), 1, fagenda); /* write struct inreg to file */
  printf("Continuati[D/N]?");temp=getch();
 }
 while(toupper(temp)!='N'); // ciclu infinit ? NU!
 fclose(fagenda); /* close file */
}

void listare(void){
 int contor=0;
 puts("\nListarea agendei:");
 mod[0]='r';
 if ((fagenda= fopen("agenda.jo", mod)) == NULL) /* open file agenda */
   fprintf(stderr, "Cannot open output file.\n");
 while(fread(&inreg, sizeof(inreg), 1, fagenda)!=0) /* write struct inreg to file */
   printf("%d: %s, %s, %s\n",++contor,inreg.nume,inreg.adresa,inreg.tel);
 fclose(fagenda); /* close file */
}

void main(void)
```

```
{
  if ((fagenda= fopen("agenda.jo", mod)) == NULL) /* open file agenda */
     fprintf(stderr, "Cannot open output file.\n");
  creare();
  listare();
}
```

## Probleme ce necesită back-tracking

Am explicat pe larg această metodă de programare într-un capitol separat. În acest capitol vom oferi doar cîteva exemple de probleme rezolvate. Majoritatea dintre ele sînt de maximă dificultate şi nu li se cunoaşte o altfel de rezolvare decît prin această metodă. Din fericire, această metodă de proiectare a soluției are un caracter foarte general şi "funcționează" în fiecare caz. Din nefericire, în practică, atunci cînd dimensiunea datelor de intrare este consistentă (avînd valori cel puțin de ordinul sutelor) programul rezultat devine, prin durata astronomică de execuție, total inutilizabil.

Atragem atenția că doar simpla lecturare a acestor exemple de probleme de back-tracking rezolvate nu permite nicidecum însuşirea acestei metode de proiectare a soluțiilor. Este necesară mai întîi implicarea şi participare personală, a celui ce-şi propune să învețe această metodă, încercînd direct soluționarea lor şi abia apoi comparînd soluția rezultată cu cea propusă de noi.

**Problema clasică de programare care necesită back-tracking (revenirea pe urma lăsată) este problema ieşirii din labirint.**

- iată o soluție simplă care inițializează labirintul în mod static, ca o matrice de caractere

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define XMAX 6
#define YMAX 6

char a[XMAX+1][YMAX+1]={
"*******",
"*    *  *",
"* *  *  *",
"*  * ****",
"**  *    *",
"*    *    ",
"********"
};
int x0=1,y0=2;

void print(void){
int i,j;
  for(i=0;i<=XMAX;i++){
   for(j=0;j<=YMAX;j++)putchar(a[i][j]);
   putchar('\n');
  }
  getchar();clrscr();
}

void escape(int x,int y){
  if(x==XMAX || y==YMAX){ puts("Succes!");exit(1);}
  a[x][y]='*';print();
  if(a[x][y+1]==' '){puts("la dreapta");escape(x,y+1);}
  if(a[x+1][y]==' '){puts("in jos    ");escape(x+1,y);}
  if(a[x][y-1]==' '){puts("la stinga ");escape(x,y-1);}
  if(a[x-1][y]==' '){puts("in sus    ");escape(x-1,y);}
  return;
}

void main(void){
 escape(x0,y0);
 puts("Traped!");
```

```
}
```

**Să se genereze toate şirurile de lungime *n* formate numai din caracterele *a, b* şi *c* a.î. să nu existe două subşiruri identice alăturate.**

- de exemplu, dacă *n=3* putem avea şiruri de forma *abc, cab, bcb,* etc. dar nu şi şiruri de forma *aab*; pentru *n=4* nu putem genera şiruri de forma *abab, baac,* etc.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define byte unsigned char

char car[4]=" abc";
unsigned int n,contor;

int Valid(char *s,char c,byte k){  // functia de validare a sirului generat
byte i,j,ok,Val=1;                 // prin concatenarea unui singur caracter
   for(i=k-1;i>=(k+1)/2;i--)
      if (s[i]==c){
        ok=1;
        for(j=1;j<=k-i-1;j++)
          if (s[i-j]!=s[k-j]){ok=0;break;}
        if (ok) { Val=0;break;}
      }
   return Val;
}

void ConcatSir(char *s,byte k){    // functia ce implementeaza back-tracking-ul
byte i;                            // in varianta recursiva
   if(k<=n){
      for(i=1;i<=3;i++)
        if (Valid(s,car[i],k)) {
          s[k]=car[i];s[k+1]='\0';
          ConcatSir(s,k+1);
        }
   } else { contor++;printf("%4i:%s",contor,s);}
}

void main(void){
   printf("n:");scanf("%i",&n);
   contor=0;
   ConcatSir(" ",1);
   exit;
}
```

**Să se afişeze toate descompunerile unei sume *s* într-un număr minim de monezi ale unui sistem monetar de *n* valori.**

- de exemplu, în cazul unui sistem monetar de forma *10, 5, 3, 1* putem descompune suma *18* în diverse moduri dar soluția minimală necesită doar 3 monezi: *18=1 x 10+1 x 5+1 x 3* ; descompunerea minimală poate să nu fie unică ; sistemul monetar trebuie să fie astfel ales încît să permită descompunerea oricărei sume începînd de la o valoare minimală în sus (orice sistem monetar conține de obicei moneda unitară *1*)

```
#include <stdio.h>
```

```
int m[10],a[10],a_final[10],s,n,nrmin=32000,kmin;

void descompune(int s,int k,int contor){
register int i;
 if(s==0)
   if(contor<nrmin){
           nrmin=contor;kmin=k-1;
           for(i=1;i<=k;i++)a_final[i]=a[i];
           for(i=k+1;i<=n;i++)a_final[i]=0;
           return;
          }
   else return;
 if(k>n) return;
 if(k==n){
   a[k]=s/m[k];descompune(s-a[k]*m[k],k+1,contor+a[k]);
 }
 else for(i=s/m[k];i>=0;i--){
                                  a[k]=i;descompune(s-i*m[k],k+1,contor+i);
       }
}

void main(void){
 int i;
 printf("Introd.nr.de valori n a sistemului monetar:");scanf("%i",&n);
 printf("Introd.in ordine descresc.cele %i valori monetare:",n);
 for(i=1;i<=n;i++)scanf("%i",&m[i]);
 printf("Introd.suma s:");scanf("%i",&s);
 descompune(s,1,0);
 if(nrmin>0) for(i=1;i<=kmin;i++)printf("%i * %i,",a_final[i],m[i]);
 else puts("Nu se poate descompune !");
}
```

**Să se afişeze toate descompunerile unui număr *s* ca sumă de *n* elemente.**

- de exemplu, pentru *s=13* şi *n=3* avem următoarele *14* descompuneri *13= 1+1+11= 1+2+10= 1+3+9=...= 1+6+6= 2+2+9= 2+3+8= 2+4+7= 2+5+6= 3+3+7= 3+4+6= 3+5+5= 4+4+5*
- deşi este cu totul altă problemă decît cea dinainte, putem observa asemănarea dintre cele două soluții (ambele sînt date în varianta recursivă)

```
#include <stdio.h>

int a[10],s,n,contor=0;

void descompune(int s,int k){
register int i;
 if(k==1){ a[n]=s;printf("%3i:",++contor);
                          for(i=1;i<=n;i++)printf("%i ",a[i]);
                          puts("");return;
                          }
 else for(i=1;i<s;i++){ a[n+1-k]=i;descompune(s-i,k-1);}
}

void main(void){
 printf("Introd.suma s si in cit se descompune n:");scanf("%i %i",&s,&n);
 descompune(s,n);
 getchar();
}
```

**Să se afişeze toate descompunerile unui număr *s* ca sumă de n elemente distincte.**

- aceasta este o variantă a problemei dinainte; puteți sesizați unde apare diferența în rezolvare ?

```
#include <stdio.h>

int a[10],s,n,contor=0;

void descompune(int s,int k){
register int i;
 if(k==0){ printf("%3i:",++contor);
                          for(i=1;i<=n;i++)printf("%4i",a[i]);
                          puts("");return;
                         }
 else for(i=a[n-k]+1;i<s;i++){ a[n+1-k]=i;descompune(s-i,k-1);}
}

void main(void){
 printf("Introd.suma s si in cit se descompune n:");scanf("%i %i",&s,&n);
 a[0]=0;
 descompune(s,n);
 if(contor==0)puts("Nu se poate descompune !");
 getchar();
}
```

## Probleme cu soluție surprinzătoare

În rezolvarea fiecăreia din problemele următoare este foarte uşor de căzut în capcana soluționării de genul "la prima mînă" sau *brute-force-approach* în limba engleză (abordare *în forță brută*). Este cea mai des întîlnită capcană pentru programatorii lipsiți de subtilitate, experiență sau cultură de specialitate. Este şi aceasta o rezolvare, desigur, dar lipsa ei de eficiență şi de eleganță este evidentă. Tocmai de aceea, considerăm foarte utilă prezentarea cîtorva exemple elocvente, împreună cu soluțiile lor. Unele dintre probleme au fost selecționate dintre problemele date la concursurile şi olimpiadele şcolare de programare .

Prin acest capitol nu urmărim doar însuşirea unor cunoştințe practice de programare ci, mai ales, aprofundarea capacității de analiză şi proiectare a soluțiilor. Aceasta presupune un salt calitativ în învățarea programării şi de aceea acest capitol devine cu adevărat util numai pentru acei programatori inteligenți şi dornici de auto-perfecționare. Sau pentru cei care se pregătesc pentru participarea la concursurile şi olimpiadele de informatică.

Soluțiile oferite de noi sînt, pentru fiecare problemă, eficiente şi "elegante". Acest fapt se datorează accentului pus pe aprofundarea şi îmbunătățirea primei analize a problemei.

Putem atunci spune, că motto-ul acestui capitol este: **"Nu te mulțumi niciodată cu soluția la prima mînă !"**.

**Să se afişeze numărul cuburilor perfecte mai mici decît n.**

**Analiza problemei - elaborarea algoritmului:**

Capcana problemei constă în tentativa de a parcurge printr-un ciclu *for* toate numerele de la *1* la *n* şi de a contoriza cele care sînt cuburi perfecte.
La o a nouă privire, mai atentă, observăm că partea întreagă a radicalului de ordinul 3 din *n* ne oferă acel număr care ridicat la a 3-a este cel mai apropiat cub de *n*. Deci, partea întreagp a radicalului de ordinul 3 din *n* este egală chiar cu numărul cuburilor mai mici decît *n.*
(Este suficient să calculăm radical de ordinul 3 din *n* pentru a afla cîte cuburi mai mici decît n există.)

```
program cuburi_perfecte;
var n,i,nr_cub:word;
BEGIN
write('n=');readln(n);
nr_cub:=trunc(exp(1/3*ln(n)));
writeln('numarul cuburilor perfecte < ',n,' este = ', nr_cub);
readln;
END.
```

**Se citesc *m, n* numărătorul şi numitorul unei fracții. Să se simplifice această fracție.**

**Analiza problemei - elaborarea algoritmului:**
Capcana constă în a efectua simplificarea pas cu pas, căutînd pe rînd fiecare divizor comun al numărătorului şi numitorului. În plus, ar trebui să avem grijă că, pentru unii divizori comuni, este nevoie de o simplificare repetată. Deci, două cicluri imbricate !
-pentru a obține o fracție ireductibilă este suficient să o simplificăm o singură dată cu *cmmdc* al numitorului şi al numărătorului,eliminîndu-se astfel simplificările succesive
-vom folosi subalgoritmul (Euclid) care calculează cmmdc al numărătorului şi al numitorului.

```
program simplificare;
var m,n:word;

function cmmdc(m,n:word):word;
begin
 while m<>n do
  if m>n then m:=m-n
```

```
  else n:=n-m;
  cmmdc:=m;
end;

BEGIN
write('numaratorul fractiei m= ');readln(m);
write('numitorul fractiei n= ');readln(n);
if n=0 then writeln('Fractie inexistenta.')
else
 if m=0 then  writeln(m,'/',n,'=',0)
 else
   writeln(m,'/',n,' = ',m div cmmdc(m,n),'/',n div cmmdc(m,n));
readln;
END.
```

**Se citesc *a, b, c* întregi. Să se determine toate perechile întregi (x,y) soluții ale ecuației ax+by=c.**

**Analiza problemei – elaborarea algoritmului;**

Problema a fost dată la olimpiada şcolară de informatică. Ea pare la prima vedere imposibilă. Există ecuații, de exemplu: 3x+2y=1 care are o infinitate de soluții …, (1,-1), (3,-4), (5,-7), (7,-10),… Cum ar putea fi afişată atunci pe ecran o mulțime infinită de perechi ? Soluția este de a afişa această mulțime printr-o descriere sintetică a ei (afişînd formula care poate genera toate perechile ce o compun).
1. daca c=1 atunci exista $(x_0,y_0)$ a.î. $ax_0+by_0=1$ doar daca [a,b]=1 ; restul solutiilor (x,y) au forma $x=x_0+kb$ , $y=y_0-ka$, cu k intreg.
2. daca c>1 atunci exista solutiile $(x_0,y_0)$ doar daca [a,b]|c; restul solutiilor se construiesc la fel;
prin [a,b] se înțelege cmmdc(a,b)
Programul trebuie doar să determine $x_0$ şi $y_0$.

```
Program ax_plus_by_egal_c;
Var a,b,c,x0,y0,y:integer;
BEGIN
  Write('a,b,c=');Readln(a,b,c);
  x0:=0;
  For y:=0 to a do
    If abs(c-b*y) mod a=0 then begin
       y0:=y;x0:=(c-b*y) div a;break;
    end;
  If x0<>0 then Writeln('Sol. (x,y) ale ec. ',a,'x+',b,'y=',c,' sint (',x0,'+k*',b,',',y0,'-k*',a,')')
  else Writeln('Nu exista solutii pentru ecuatia ',a,'x+',b,'y=',c);
END.
```

```
/*Varianta C de solutionare:
1. daca c=1 atunci exista (x0,y0) a.i. ax0+by0=1 doar daca cmmdc[a,b]=1 ;
restul solutiilor (x,y) au forma x=x0+kb y=y0-ka, cu k intreg.
2. daca c>1 atunci exista solutiile (x0,y0) doar daca cmmdc[a,b] | c;
restul solutiilor se construiesc la fel.
3. exista posibilitatea ca, pornind de la perechi (x0,y0) distincte, sa se
obtina solutii noi diferite (multimi infinite de perechi distincte).
4. toate solutiile (multimi infinite de perechi) pleaca de la o pereche
(x0,y0) aflata in dreptunghiul (-b,-a)x(b,a).
Un bun exemplu este ecuatia 18x+42y=6.*/

#include <stdio.h>
#include <math.h>

int a,b,c,x0=0,y0=0,y,k;

void main(void){
```

```
 printf("a,b,c:");scanf("%i %i %i",&a,&b,&c);

 printf("Ecuatia %ix+%iy=%i are sol.de forma:",a,b,c);
 for(y=0;y<=a;y++)
  if(abs(c-b*y) % a==0){
    y0=y;x0=(c-b*y) / a;
          if(x0!=0){
            printf("\n %i*k%+i, -(%i*k-%i), de ex. ",b,x0,a,y0);
            for(k=-2;k<=2;k++)printf("(%i,%i) ",x0+k*b,y0-k*a);
    }
  }
 if(!x0 && !y0 && c)printf("Nu exista solutii pentru ecuatia %ix+%iy=%i",a,b,c);
}
```

**Se citeşte *n* o valoare întreagă pozitivă. Să se determine toate descompunerile în diferență de pătrate ale lui n.**

**Analiza problemei – elaborarea algoritmului:**
Arătăm în continuare cum se poate evita soluția "banală"-capcană ce-ar consta în două cicluri *for* imbricate. Soluția următoare efectuează doar *radical din n* paşi, față de $n^2$ paşi ai soluției "la prima mînă".
-pentru a determina toate descompunerile in diferenta de patrate ale lui n pornim de la formula $a^2-b^2=(a+b)(a-b)=n$
-observam ca produsul termenilor a+b si a-b este produsul a doi dintre divizorii lui *n*,unul din termeni este divizor (d) a lui *n* celalalt este tot divizor a lui *n* si il aflam impartindu-l pe *n* la d (*n* div x)
-notam cu x primul divizor a lui *n* (x=d) si cu y=*n* div x si obtinem relatiile
  a+b=x deci un sistem de doua ecuatii cu doua necunoscute ,pe care il rezolvam
  a-b=y prin metoda reducerii ,si avem 2a=x+y => a=(x+y )/2 , b=(y-x)/2,
-pentru ca (x+y)/2 sa fie o solutie a ecuatiei $a^2-b^2=(a+b)(a-b)=n$ trebuie ca x+y sa fie un numar par si y-x sa fie un numar par
-daca aceasta conditie este indeplinita afisam solutia care indeplineste conditia ceruta.

```
Program descompunere_patrate;
var n,d,a,b,x,y:integer;
BEGIN
write('n=');readln(n);
for d:=1 to  trunc(sqrt(n)) do
 if n mod d =0 then
   begin
     x:=d;
     y:=n div x;
     if (x+y) mod 2 =0 then
       begin
        a:=(x+y) div 2;
        b:=(y-x) div 2;
        writeln(n,'=',a*a,'-',b*b);
       end;
   end;
readln;
END.
```

**Se citeşte *n* şi $x_1, x_2, \ldots, x_n$ rădăcinile întregi ale unui polinom de gradul *n*. Se cere să se determine pe baza relațiilor lui Viete coeficienții $a_n, a_{n-1}, \ldots, a_1, a_0$.**

**Analiza problemei – elaborarea algoritmului;**

Cea mai des întîlnită rezolvare este cea de tip back-tracking, aparent mai uşoară, dar în fapt extrem de ineficientă pentru *n* nu mare ci doar măricel ! Următoarea soluție de tip iterativ este o mică "bijuterie" de program iterativ şi de aceea vă lăsăm plăcerea de a-l înțelege singuri.

```
#include <stdio.h>

void main(void){
 int a[100],x[100],n,i,k;
 printf("n=");scanf("%d",&n);
 printf("Radacinile:\n");
 for(i=0;i<n;i++){
  printf("x[%d]=",i);scanf("%d",&x[i]);a[i]=0;
 }
 a[0]=1;a[n]=0;
 for(k=1;k<=n;k++){
  for(i=k;i>0;i--)
            a[i]=a[i-1]-a[i]*x[k-1];
  a[0]*=-x[k-1];
 }
 for(i=n;i>=0;i--) printf("a[%d]=%d ",i,a[i]);
}
```

**Se citeşte *n*. Să se afişeze toate numerele de *n* cifre, formate numai cu cifrele *1* şi *2*, divizibile cu $2^n$.**

**Analiza problemei – elaborarea algoritmului:**

Problema a fost dată la olimpiada şcolară de informatică. Abordarea "în forță" a acestei probleme nu duce la nici un rezultat:

> daca s-ar alege varianta de rezolvare "la prima mina" ar trebui verificate toate cele $2^n$ posibilitati, adica toate cele $2^n$ numere de n cifre ce se pot forma numai cu cifrele 1 si 2 (cite 2 posibilitati pentru fiecare pozitie). In acest caz, programul avind o complexitate exponentiala, ar dura un timp exponential, pt. n=50 ar dura cît vîrsta sistemului nostru solar !

pt.n=1 avem unica solutie numarul 2;
pt. n=2 avem deasemenea unica solutie 12; observam ca 2-ul este "folosit"
pt. n=3 avem deasemenea unica solutie 112; observam ca 12 este din nou "folosit"
In general, se deduce ca numerele de n cifre, ce trebuie sa se divida cu $2^n$ , se divid cu $2^{n-1}$; ele se pot scrie sub forma $c*10^{(n-1)}+M=c*2^{n-1}*5^{n-1}+M=$ Multiplu($2^{n-1}$)+M; inseamna ca M (cele n-1 cifre ramase) trebuie sa se divida cu $2^{n-1}$; inseamna ca M este unul din numerele gasite ca solutie la pasul n-1.
Daca exista o singura solutie M pt.cazul n-1 (M se divide cu $2^{n-1}$) acest nr.se poate scrie $M=2^{(n-1)}*$par sau $2^{(n-1)}*$impar, rezulta ca M mod $2^n=0$ sau M mod $2^n=2^{(n-1)}$. Deci,in cazul a n cifre din cele doua posibilitati (1M sau 2M) se va alege astfel unica solutie:
 daca M mod $2^n=0$ atunci solutia este $2M=2*10^{(n-1)}+M=$Multiplu($2^n$)
 daca M mod $2^n=2^{(n-1)}$ atunci solutia este $1M=10^{(n-1)}+M=2(^{(n-1)}*5^{(n1)}+M=$Multiplu($2^n$)!
Solutia propusa este una iterativa şi face **maximum n paşi** !

```
Program 1_2_si_2_la_n;
Var
  nr,zece_la_n:longint;
  n,i:byte;
BEGIN
  Writeln('Se citeste n. Sa se afiseze toate numerele de n cifre,');
  Writeln('formate numai cu cifrele 1 si 2, si divizibile cu 2^n.');
  Write('Introduceti n (max.10):');Readln(n);
  nr:=2;zece_la_n:=1;
  For i:=2 to n do begin
    zece_la_n:=10*zece_la_n;
    If nr mod (1 shl i)=0 then nr:=2*zece_la_n+nr
    else nr:=zece_la_n+nr;
  end;
  Writeln('Solutia este:',nr);
readln;
END.
```

**Se citeşte *n*. Să se determine o alegere a semnelor + şi - astfel încît să avem relația ±1±2±...±n=0.**

**Analiza problemei – elaborarea algoritmului:**

Problema a fost dată la olimpiada şcolară de informatică. Daca se incearca o abordare "in forta" si "la prima mina" vor trebui verificate $2^n$ posibilitati de asezare a semnelor + si -. Adica se obtine un algoritm exponential, total ineficient. Soluția "elegantă" ce rezultă printr-o analiză mai aprofundată:

-mai intai se va imparti suma in doua parti: cea cu plus si cea cu minus.

Privindu-se atent se observa ca se pot deosebi trei cazuri:

1. cind avem intre cele n numere un numar impar de numere impare (de ex.n=3,5,6...) caz in care numerele impare nu pot fi repartizate in cele doua parti (plus si minus) decit astfel: un nr.par de numere impare intr-o parte si un nr.impar de nr impare in cealalta; implica ca cele doua parti au paritati diferite, deci suma lor nu poate fi 0 !

Acest caz apare cind n=4k+1, 4k+2.

2. cind n=4k atunci numerele de la 1 la n pot fi grupate cite patru astfel:

1-2-3+4, ..., (4i+1)-(4i+2)-(4i+3)+(4i+4), ... si vor avea suma 0 pe fiecare grupa de patru !

3. altfel, n=4k+3, putem grupa numerele asemanator ca la cazul dinainte cu exceptia primei grupe: -1-2+3, 4-5-6+7, ..., (4i)-(4i+1)-(4i+2)+(4i+3),...reazultind din nou suma 0 pe fiecare grupa !

```
Program Plus_Minus;
Var
  n,i,c:byte;
BEGIN
   Writeln('Se citeste n. Sa se determine o alegere a semnelor + si - ');
   Writeln('astfel incit sa avem relatia ±1±2±...±n=0.');
   Write('n:');Readln(n);
   c:=n mod 4;
   case c of
   1,2: Writeln('Nu exista solutie.');
   0: For i:=1 to n div 4 do
       write('+',4*(i-1)+1,'-',4*(i-1)+2,'-',4*(i-1)+3,'+',4*(i-1)+4);
   3:begin
      Write('-1-2+3');
      For i:=1 to n div 4 do
       write('+',4*i,'-',4*i+1,'-',4*i+2,'+',4*i+3);
     end;
   end;
   Readln;
END.
```

## Elemente de programare a PC - urilor

Oferim în continuare cîteva exemple de programe, unele în Pascal, altele în C, pentru a permite celor pasionați să-şi însuşească cunoştinţele minimale de programare a PC-urilor: lucrul cu tastatura, accesul direct la memorie, lucrul în modul grafic, etc. Pentru cei ce doresc să aprofundeze acest subiect sau doresc cît mai multe detalii le recomandăm, pe lîngă citirea atentă a help-ului Turbo Pascal-ului sau a Turbo C-ului, folosirea utilitarului **TechHelp** specializat în descrierea programării PC-urilor**.**

Ideea care ar defini cel mai bine acest tip de cunoştinţe de programare este conţinută în cunoscuta expresie : "Secrete mici, efecte mari !".

**// Un simplu program muzical**

```
#include <stdio.h>
#include <dos.h>
#include <conio.h>
main(){    /* Do   do# Re   re# Mi   Fa   fa# sOl sol# La   la#   Si */
int octava[]={65 , 69 , 73 , 78 , 82 , 87 , 92 , 98 , 104 , 110 , 116 , 123};
int i,j,nr_octava,i_nota,timp=500;
float masura,durata,durata_masura;
char *linia="42$2R2R4M4F2O2L1R2R2S2S4L4O2O2"; //$4D2D4$3S4L2";

do{
 masura=(float)(linia[0]-'0')/(linia[1]-'0');durata_masura=0;
 for(i=2;linia[i]!='\0';i++){
  if (i%2==0){
    switch(linia[i]){
      case '$' : {nr_octava=1;for(j=linia[++i]-'0';j>0;j--)nr_octava*=2;}
                        break;
      case 'D' : i_nota=0;break;
      case 'd' : i_nota=1;break;
      case 'R' : i_nota=2;break;
      case 'r' : i_nota=3;break;
      case 'M' : i_nota=4;break;
      case 'F' : i_nota=5;break;
      case 'f' : i_nota=6;break;
      case 'O' : i_nota=7;break;
      case 'o' : i_nota=8;break;
      case 'L' : i_nota=9;break;
      case 'l' : i_nota=10;break;
      case 'S' : i_nota=11;break;
    }
  } else {
    if (linia[i]=='6') durata=1/16; else durata=1/(float)(linia[i]-'0');
    durata_masura+=durata;
    if (durata_masura>masura) { nosound();durata_masura=0;}
    sound(nr_octava*octava[i_nota]);
    delay(durata*timp);
  }          /* else */
 }           /* for */
}            /* do */
while (!kbhit());
nosound();
}
```

**Program Citite_Taste;**
uses crt;

```
var c:char;
     shift:byte absolute $40:$17; { adresa octetului de stare a tastaturii }
begin
repeat
    c:=readkey;
    if (shift and $3>0) then
           write(' shift ',c,':',Ord(c))
    else write(' ',c,':',Ord(c));
until c=#27;
end.
```

**// Program C pt. afisarea Tabelului codurilor ASCII;**

```
#include <stdio.h>

void main(){
unsigned short c;
for(c=0;c<=255;c++)
    switch(c){
     case 7  : printf("b%3uł",c);break; // beep
     case 8  : printf("B%3uł",c);break; // back space
     case 9  : printf("T%3uł",c);break; // tab
     case 10 : printf("L%3uł",c);break; // line feed
     case 13 : printf("R%3uł",c);break; // return
     case 27 : printf("E%3uł",c);break; // escape
     default : printf("%c%3uł",c,c); // caractere afisabile
    };
}
```

**Program Tenis;**

```
{ Joc demonstrativ al posibilitatilor de folosire a accesului direct
la memoria ecran. Paletele sint actionate de tastele 'A' si 'W', respectiv
'sageata sus' si 'sageata jos'. }
Uses Crt;
Const viteza=1500;
Type Ecran=Record
        car:char;
        atrib:byte;
     End;
Var
  scr:array[1..25,1..80] of Ecran absolute $b800:$0; { Adresa de memoriei ecran in mod text }
  x,y,x0,y0:byte;
  i,d,s:integer;
  u:real;
  ok:boolean;
  tasta:char;
  yP1:array[1..5]of byte;
  yP2:array[1..5]of byte;
  uP:array[1..5]of real;

Procedure Paleta1(tip:char);
Begin                    {generare paleta 1}
    for i:=1 to 5 do
       scr[yP1[i],76].car:=tip;
end;
Procedure Paleta2(tip:char);
Begin                    {generare paleta 2}
   for i:=1 to 5 do
```

```
        scr[yP2[i],5].car:=tip;
End;
Procedure Mutapaleta1;
Begin
  Paleta1(' ');
  if (tasta=#80) and (yP1[i]<24) then          {miscarea paletei 1}
    for i:=1 to 5 do Inc(yP1[i]);
  if (tasta=#72) and (yP1[i]>6) then
    for i:=1 to 5 do Dec(yP1[i]);
End;
Procedure Mutapaleta2;
Begin
  Paleta2(' ');                          {miscarea paletei 2}
    if (tasta=#122) and (yP2[i]<24) then
    for i:=1 to 5 do Inc(yP2[i]);
  if (tasta=#119) and (yP2[i]>6) then
    for i:=1 to 5 do Dec(yP2[i]);
End;
procedure cantec; {genereaza cantecul final}
begin sound(400);delay(800);
     sound(500);delay(800);
     sound(600);delay(800);
     sound(650);delay(800);
     sound(600);delay(800);
     sound(700);delay(800);
     sound(650);delay(1000);
end;
Begin                          {program principal-generare cadru}
   Clrscr;
   d:=0;s:=0;
{   writeln('________ ________ _______ ______   _________ ');
   write(char(179),'        ',char(179),'      ',char(179),'     ');
   writeln(char(179),'     ',char(179));
   readln;}
   clrscr;
   For x:=1 to 80 do begin
      scr[1,x].car :=#219;
      scr[25,x].car:=#219;
   end;
   For y:=2 to 9 do begin   {poarta}
      scr[y,1].car :=#219;
      scr[y,80].car:=#219;
   end;
   For y:=17 to 24 do begin
      scr[y,1].car :=#219;
      scr[y,80].car:=#219;
   end;
   x0:=40;
   y0:=13;
   u:=20*PI/180;     {initializare miscare minge}
   x:=x0;
   y:=y0;
   for i:=1 to 5 do begin
      yP1[i]:=10+i;
      yP2[i]:=10+i;
      uP[i]:=(i/3*PI-PI)/15; {unghiul de dispersie a paletei}
   end;
   tasta:=' ';
   repeat                          {miscare minge}
       if ((u>=0) and (u<PI/2) or (u > 3*PI/2) and (u<2*PI)) then inc(x)
```

```
else dec(x);
y:=y0+Trunc(Abs(x-x0) * Sin(u)/Cos(u));
if scr[y,x].car<>' ' then begin
   if (y=1)or(y=25) then begin              {ciocniri}
     u:=2*PI-u;x0:=x;
     if y=1 then y0:=2 else y0:=24;
   end;                                {-de pereti}
   if (x=1)or(x=80) then begin
     u:=PI+u;if u>2*Pi then u:=u-2*PI;
     y0:=y;
     if x=1 then x0:=2 else x0:=79;
   end;
   if x=76 then begin                  {-de palete}
    for i:=1 to 5 do
     if y=yP1[i] then begin
       sound(1000);
       u:=PI+u+uP[i];
       if u>2*Pi then u:=u-2*PI;
       x0:=x;y0:=y;
     end;
    nosound;
   end;
   if x=5 then begin                   {-de palete}
    for i:=1 to 5 do
     if y=yP2[i] then begin
       sound(600);
       u:=PI+u+uP[i];
       if u>2*Pi then u:=u-2*PI;
       x0:=x;y0:=y;
     end;
    nosound;
   end;
end
else if not (((x=1)or(x=80)) and((y<17)and(y>8))) then
  begin                {gol}
    scr[y,x].car:='0';
    i:=1;
    ok:=false;
    repeat
        ok:=keypressed;
        inc(i);
    until (i=viteza)or ok;
    if ok then begin
      tasta:=readkey;
      if tasta = #0 then tasta:=readkey;
      mutapaleta1;
      mutapaleta2;
    end;
    Paleta1(#219);
    Paleta2(#219);
    scr[y,x].car:=' ';
    scr[y,x].car:=' ';
end
else begin
       sound(800);
       if (x>=80)and(y>9)and(y<17) then d:=d+1;
       if (x<=1)and(y>9)and(y<17) then s:=s+1;
       textcolor(2);
       textbackground(7);
       gotoxy(39,2);
```

```
            write('SCOR');
            gotoxy(38,3);
            write(' ',d,' : ',s);
            if (d=5)or(s=5) then begin
                        gotoxy(35,10);
                        write(' G A M E  O V E R ');
                        cantec; nosound;
                        halt;
                        end;
            delay(1500);
            paleta1(' ');
            paleta2(' ');
            x0:=40;
            y0:=13;
            u:=20*PI/180;     {reinitializare miscare minge}
            x:=x0;
            y:=y0;
            for i:=1 to 5 do begin
            yP1[i]:=10+i;
            yP2[i]:=10+i;
            uP[i]:=(i/3*PI-PI)/5;
            end;
            tasta:=' ';
            nosound;
        end;
   until tasta=#27;
End.
```

**Program Biliard; { demonstrativ pentru folosirea modului grafic }**

```
uses Graph,Crt;
Const nr_obiecte=10;
     raza=25;
     pasx=3;pasy=2;
     viteza=10; { de la 0 la 10 }
var
 grDriver,grMode,ErrCode: Integer;
 i,xMax,yMax,xtmp,ytmp:word;
 x,y:Array[1..nr_obiecte] of word;
 sensx,sensy:Array[1..nr_obiecte] of shortint;

Procedure Deseneaza(x,y,color:word);
Const bucati=12;
Var x1,y1,unghi,Xasp,Yasp:word;
Begin
  SetWriteMode(XORPut);SetColor(color);
  GetAspectRatio(Xasp, Yasp);
  unghi:=0;
  x1:=x+Trunc(raza*cos(unghi*2*PI/bucati));
  y1:=y+Trunc(raza*sin(unghi*2*PI/bucati)*Xasp/Yasp);
  For unghi:=1 to bucati do begin
     xtmp:=x+Trunc(raza*cos(unghi*2*PI/bucati));
     ytmp:=y+Trunc(raza*sin(unghi*2*PI/bucati)*Xasp/Yasp);
     Line(x1,y1,xtmp,ytmp);Line(x,y,x1,y1);
     x1:=xtmp;y1:=ytmp;
  end;
End;

begin
 grDriver := Detect;
```

```
 InitGraph(grDriver, grMode,'');
 ErrCode := GraphResult;
 if ErrCode = grOk then
 begin  { Do graphics }
   xMax:=GetMaxX;yMax:=GetMaxY;
   Rectangle(0,0,xMax,yMax);
   Randomize;
   For i:=1 to nr_obiecte do begin
      x[i]:=raza+Random(xMax-2*raza);y[i]:=raza+Random(yMax-2*raza);
      sensx[i]:=-1+(i mod 2)*2;sensy[i]:=-sensx[i];
      Deseneaza(x[i],y[i],i);
   end;
   Repeat
     For i:=1 to nr_obiecte do begin
        Deseneaza(x[i],y[i],i);
        xtmp:=x[i]+pasx*sensx[i];ytmp:=y[i]+pasy*sensy[i];
        If (xtmp>raza) and (xtmp<xMax-raza) then x[i]:=xtmp
        else sensx[i]:=-sensx[i];
        If (ytmp>raza) and (ytmp<yMax-raza) then y[i]:=ytmp
        else sensy[i]:=-sensy[i];
        Deseneaza(x[i],y[i],i);
        Delay(100-10*viteza);
     end;
   Until KeyPressed;
   Readln;
   CloseGraph;
 end
 else
   Writeln('Graphics error:', GraphErrorMsg(ErrCode));
end.
```

**// Program C de umplere a ecranului text prin acces direct la memoria ecran**

```
#include <dos.h>
#include <conio.h>

struct scrcar{
  unsigned char car,atrib;
} far *ecran;

int lin,col;
int culoare=BLUE,fundal=LIGHTGRAY;

void main(void){
ecran=(struct scrcar far *)MK_FP(0xb800,0);
for(lin=0;lin<25;lin++)
for(col=0;col<80;col++) {
            ecran[lin*80+col].car='*';
            ecran[lin*80+col].atrib=fundal*16+culoare;
}
getch();
}
```

**Program Acces_direct_ecran_grafic320_200;**

```
{ Fiecare jumatate de ecran se genereaza din cealalta jumatate
pe baza proprietatilor automatelor celulare – asemanator ca in jocul Life }
Uses crt;
Const maxl=200-1;
```

```
     maxc=320-1;
     mijl=maxc div 2;
Type Matrice=array[0..maxl,0..maxc] of byte;
var
  scr:Matrice absolute $A000:0; { adresa memoriei ecran in modul grafic 320x200 }
  i,j,k,l,c,x:integer;
  ok:char;
BEGIN
asm         {initializeaza in mod grafic 320x200x250 NU in 640x400x256}
  mov ah,0
  mov al,13h
  int 10h;
end;
randomize;x:=random(maxc);
for k:=1 to 2 do
 for i:=0 to maxl do
 for j:=0 to mijl do
    scr[i,j+k*mijl]:=random(maxc) ;
k:=0;
repeat
repeat
     for i:=0 to maxl do
     for j:=0 to mijl do begin
      l:=i;c:=j+k*mijl;
      if (scr[(l-1)mod maxl,c]<scr[l,c])and
        (scr[l,(c-1)mod mijl]<scr[l,c]) then
          scr[i,j+((k+1)mod 2)*mijl]:=(scr[(l-1)mod maxl,c]+scr[l,(c-1)mod mijl]+ x)div 3-1
      else if (scr[l,(c+1)mod mijl]>scr[l,c])and
            (scr[(l+1)mod maxl,c]>scr[l,c]) then
              scr[i,j+((k+1)mod 2)*mijl]:=(scr[(l+1)mod maxl,c]+scr[l,(c+1)mod mijl]+ x) div 3+1
          else scr[i,j+((k+1)mod 2)*mijl]:=scr[l,c]+1;
     end;
     k:=(k+1) mod 2;
until keypressed;
ok:=readkey;x:=random(maxc);
if ok<>#27 then ok:=readkey;
until ok=#27;
{readln;}
asm         {inchide modul grafic}
    mov ax,0
    int 10h
end;
END.
```

**Program Mouse; { Gestionarea mouse-ului prin apelul intreruperii de sistem $33 }**

```
uses Crt,Graph,Dos;
var
 grDriver,grMode,ErrCode  : Integer;
 mfunc,buton,mx,my,xf,yf,x,y:word;
 xi,yi:integer;
 s1,s2,s3:string[5];
 P      : pointer;
 Size   : Word;

{    Intr $33, nr.fctiei dorite in AX:
       00 mouse reset
       01 cuplare cursor mouse (vizibil)
       02 decuplare cursor mouse(ascuns)
       03 determ.unei apasari pe tasta si semnalare pozitie
```

```
        04 pozitionarea cursorului de mouse
        05 inform.suplim.despre apasarea tastelor
        06 inreg.tastelor de mouse eliberate
        07 stabilire domeniu orizontal(minim si maxim)
        08 - || -    - || -vertical - || - - || -
        09 selectare cursor grafic
        10 selectare cursor text
        13/14 emulare creion optic conectat/deconectat
        15 stabilire sensibilitate mouse
        29 fixarea paginii ecran in care mouse-ul e vizibil
        30 afisarea - || - - || - - || - - || -

procedure MouseReg;
var reg:registers;
begin
    reg.ax:=mfunc;reg.bx:=buton;reg.cx:=mx;reg.dx:=my;
    intr($33,reg);
    mfunc:=reg.ax;buton:=reg.bx;mx:=reg.cx;my:=reg.dx;
end;
}
procedure MouseAsm;ASSEMBLER;
    ASM
    MOV AX,mfunc
    MOV BX,buton
    MOV CX,mx
    MOV DX,my
    INT $33
    MOV mfunc,AX
    MOV buton,BX
    MOV mx,CX
    MOV my,DX
end;

Begin
  grDriver := Detect;
  InitGraph(grDriver,grMode,'');
  ErrCode := GraphResult;
  if ErrCode = grOk then
    begin
      if mem[memW[0:$cc+2]:memW[0:$cc]]=$cf then
      begin
          outtext('Mouse-ul nu este instalat!');
          readln;closegraph;halt;
      end;
      mfunc:=0;mouseasm;      {initializare}
      mfunc:=1;mouseasm;      {vizibil}
      mfunc:=3;
      mouseasm;xi:=mx;yi:=my;
      setactivepage(1);
      rectangle(xi,yi,mx,my);
      Size := ImageSize(xi,yi,mx,my);
      GetMem(P, Size); { Get memory from heap }
      GetImage(xi,yi,mx,my,P^);
      putimage(xi,yi,P^,XORput);
      setactivepage(0);
      PutImage(100, 100, P^, ORPut);
      repeat
          mouseasm;
          xi:=mx;yi:=my;
          while buton=1 do
```

```
        begin
            PutImage(100, 100, P^,XORPut);
            mouseasm;
            setactivepage(1);
            rectangle(xi,yi,mx,my);
            Size := ImageSize(xi,yi,mx,my);
            GetMem(P, Size); { Get memory from heap }
            GetImage(xi,yi,mx,my,P^);
            putimage(xi,yi,P^,XORput);
            setactivepage(0);
            PutImage(100, 100, P^, ORPut);
        end;
    until keypressed;
    mfunc:=2;mouseasm;       { decuplare mouse }
    CloseGraph;
  end
 else
  WriteLn('Graphics error:',GraphErrorMsg(ErrCode));
end.
```

**// Program C de generare a efectului grafic-plasma-prin utilizarea unor functii ale modului grafic**

```
#include <graphics.h>
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <math.h>
#include <dos.h>

int MX,MY;
int p1,p2,p3,p4,r1,r2,r3,r4;

void plasma(int x1,int x2,int y1,int y2){
 if(x2-x1<2) return;
 p1=getpixel(x1,y1);
 p2=getpixel(x2,y1);
 p3=getpixel(x2,y2);
 p4=getpixel(x1,y2);
 r1=random(4);
 r2=random(4);
 r3=random(4);
 r4=random(4);
 if (getpixel(x1+(x2-x1)/2,y1)==0) putpixel(x1+(x2-x1)/2,y1,(p1+p2)/2+r1);
 if (getpixel(x2,y1+(y2-y1)/2)==0) putpixel(x2,y1+(y2-y1)/2,(p2+p3)/2+r2);
 if (getpixel(x1+(x2-x1)/2,y2)==0) putpixel(x1+(x2-x1)/2,y2,(p3+p4)/2+r3);
 if (getpixel(x1,y1+(y2-y1)/2)==0) putpixel(x1,y1+(y2-y1)/2,(p4+p1)/2+r4);
 putpixel(x1+(x2-x1)/2,y1+(y2-y1)/2,(p1+p2+p3+p4)/4+random(2));
 plasma(x1,x1+(x2-x1)/2,y1,y1+(y2-y1)/2);
 plasma(x1+(x2-x1)/2,x2,y1,y1+(y2-y1)/2);
 plasma(x1,x1+(x2-x1)/2,y1+(y2-y1)/2,y2);
 plasma(x1+(x2-x1)/2,x2,y1+(y2-y1)/2,y2);
}

int gdriver = VGA, gmode = VGAHI, errorcode,i;
double red=20,green=30,blue=40;

struct palettetype pal;

void main(void){
/* select a driver and mode that supports the use */
```

```
/* of the setrgbpalette function.                */

/* initialize graphics and local variables */
initgraph(&gdriver, &gmode, "");

/* read result of initialization */
errorcode = graphresult();
if (errorcode != grOk)  /* an error occurred */
{
   printf("Graphics error: %s\n", grapherrormsg(errorcode));
   printf("Press any key to halt:");
   getch();
   exit(1); /* terminate with an error code */
}

/* grab a copy of the palette */
getpalette(&pal);

for (i=0; i<pal.size; i++)
            setrgbpalette(pal.colors[i], red+i, green+i, blue+i);

 randomize();
 MX=getmaxx();MY=getmaxy();
 putpixel(0,0,MAXCOLORS/2);
 putpixel(0,MY,MAXCOLORS/2);
 putpixel(MX,0,MAXCOLORS/2);
 putpixel(MX,MY,MAXCOLORS/2);
 plasma(0,MX,0,MY);
 // rotate palette
   while(!kbhit()){
     for(i=0;i<pal.size;i++)
               setrgbpalette(pal.colors[i],(int) red+i, (int) green+i, (int) blue+i);
     red+=0.5; green+=1; blue+=1.5;
   }
 closegraph();
}
```

**Program Sarpe;**

```
{ Program de joc demonstrativ: "Sarpele" culegator de numere. El este dirijat
cu ajutorul sagetilor, viteza sa de miscare poate fi modificata corespunzator
in orice moment folosind tastele de la 1 la 9. }
Uses Crt;
Const
     sc=#219;
     lungmax=95;
     maxnext=10;
     xlimit=[1,80];
     ylimit=[1,25];
Var
   sx,sy:array[1..95] of byte;
   c:char;
   i,primul,ultimul,next,tdelay,idelay:integer;
   xnext,ynext:byte;
Begin
clrscr;
randomize;
for i:=1 to 79 do begin gotoxy(i,1);write(sc);gotoxy(i,25);write(sc);end;
for i:=1 to 24 do begin gotoxy(1,i);write(sc);gotoxy(80,i);write(sc);end;
```

```
primul:=2;ultimul:=1;
for i:=primul downto ultimul do begin sx[i]:=40;sy[i]:=13;end;
next:=0;idelay:=100;
for i:=primul downto ultimul do begin
   gotoxy(sx[i],sy[i]);write(sc);
end;
c:=readkey;
while next<maxnext do
begin
 xnext:=2+random(78);ynext:=2+random(23);
 inc(next);gotoxy(xnext,ynext);write(next);
 repeat
  if keypressed then begin
    c:=readkey;tdelay:=idelay;
    if c=#0 then c:=readkey;
  end
  else tdelay:=tdelay*97 div 100;
  case c of
  '1'..'9':
        idelay:=100+100 div (ord(c)-ord('1')+1);
  #75: { stinga }
  begin
       gotoxy(sx[ultimul],sy[ultimul]);write(' ');
       if primul=lungmax then begin
         sx[1]:=sx[primul]-1;sy[1]:=sy[primul];
         primul:=1
       end
       else begin
          inc(primul);
          sx[primul]:=sx[primul-1]-1;sy[primul]:=sy[primul-1];
       end;
       if ultimul=lungmax then ultimul:=1
       else inc(ultimul);
  end;
  #77: { dreapta }
  begin
       gotoxy(sx[ultimul],sy[ultimul]);write(' ');
       if primul=lungmax then begin
         sx[1]:=sx[primul]+1;sy[1]:=sy[primul];
         primul:=1
       end
       else begin
          inc(primul);
          sx[primul]:=sx[primul-1]+1;sy[primul]:=sy[primul-1];
       end;
       if ultimul=lungmax then ultimul:=1
       else inc(ultimul);
  end;
  #72: { sus }
  begin
       gotoxy(sx[ultimul],sy[ultimul]);write(' ');
       if primul=lungmax then begin
         sx[1]:=sx[primul];sy[1]:=sy[primul]-1;
         primul:=1
       end
       else begin
          inc(primul);
          sx[primul]:=sx[primul-1];sy[primul]:=sy[primul-1]-1;
       end;
       if ultimul=lungmax then ultimul:=1
```

```
          else inc(ultimul);
    end;
    #80: { jos }
    begin
          gotoxy(sx[ultimul],sy[ultimul]);write(' ');
          if primul=lungmax then begin
            sx[1]:=sx[primul];sy[1]:=sy[primul]+1;
            primul:=1
          end
          else begin
             inc(primul);
             sx[primul]:=sx[primul-1];sy[primul]:=sy[primul-1]+1;
          end;
          if ultimul=lungmax then ultimul:=1
          else inc(ultimul);
    end;
    end;
    if primul > ultimul then
      for i:=primul downto ultimul do begin
         gotoxy(sx[i],sy[i]);write(sc);
         if (sx[primul]=sx[i]) and (sy[primul]=sy[i]) and (i<>primul) then
           c:=#27;
      end
    else
    begin
      for i:=ultimul to lungmax do begin
         gotoxy(sx[i],sy[i]);write(sc);
         if (sx[primul]=sx[i]) and (sy[primul]=sy[i]) and (i<>primul) then
           c:=#27;
      end;
      for i:=1 to primul do begin
         gotoxy(sx[i],sy[i]);write(sc);
         if (sx[primul]=sx[i]) and (sy[primul]=sy[i]) and (i<>primul) then
           c:=#27;
      end;
    end;
    if (sx[primul] in xlimit)or(sy[primul] in ylimit) then c:=#27;
    delay(tdelay);
  until (c=#27) or ((sx[primul]=xnext)and(sy[primul]=ynext));
  sound(next*30);
  if c=#27 then next:=maxnext
  else
  if ultimul-next <= 0 then begin
      for i:=lungmax+ultimul-next to lungmax do begin
        sx[i]:=sx[ultimul];sy[i]:=sy[ultimul];
      end;
      for i:=1 to ultimul do begin
        sx[i]:=sx[ultimul];sy[i]:=sy[ultimul];
      end;
      ultimul:=lungmax+ultimul-next;
  end
  else begin
      for i:=ultimul-next to ultimul do begin
        sx[i]:=sx[ultimul];sy[i]:=sy[ultimul];
      end;
      ultimul:=ultimul-next;
  end;
  delay(tdelay);
  nosound;
end; { next < maxnext}
```

```
End.
```

**Program Scan_Taste;**

```
{ Program ce demonstreaza posibilitatea de acces la codurile de scanare
ale tastaturii. Este indicat sa fie lansat in mod DOS si nu de sub Windows. }
Uses Crt,Dos;
Var
  tasta:byte;
  KbdIntVec:procedure;
{$F+}
Procedure KeyClick; interrupt;
begin
   Port[$20]:=$20; { resetarea portului de acces al tastaturii }
end;

Begin
   GetIntVec($9,@KbdIntVec); { modificarea intreruperii de tastatura }
   SetIntVec($9,Addr(KeyClick)); { cu o procedura proprie "inofensiva" }
   tasta:=0;
   repeat
      repeat until tasta<>Port[$60];
      tasta:=Port[$60];
      gotoxy(20,2);write(tasta:3);
   until tasta=129;
   SetIntVec($9,@KbdIntVec);
End.
```

**Program Taste_muzicale_V2;**

```
{ Program demonstrativ de folosire muzicala a tastaturii pe post de "orga".
Pentru o mai buna intelegere este utila consultarea programului scantast.pas }
Uses Crt,Dos;
Const
   Nota_Do:array[1..4] of integer=(33,66,132,264);
   Raport:array[1..10]of real=(24/24,27/24,30/24,32/24,36/24,40/24,45/24,
                        48/24,51/24,54/24);
   Nota:array[1..10]of string[3]=('Do','Re','Mi','Fa','Sol','La','Si',
                          'Do','Re','Mi');
   CodT:array[1..4]of byte=(44,30,16,2);

Type Pixel=Record
       atrib:byte;
       car:char;
   end;
Var
  tasta:byte;i:integer;
  KbdIntVec:procedure;
  ecran:array[1..25,1..80]of Pixel absolute $b800:0000;
{$F+}
Procedure KeyClick; interrupt;
begin
   Port[$20]:=$20;
end;
Begin
 ClrScr;
 GetIntVec($9,@KbdIntVec);
 SetIntVec($9,Addr(KeyClick));
 tasta:=0;
 repeat
```

```
   repeat until tasta<>Port[$60];
   tasta:=Port[$60];
   if (tasta>=CodT[1])and(tasta<CodT[1]+10) then
   begin
    gotoxy(5*(tasta+1-CodT[1]),24);write(Nota[tasta+1-CodT[1]]);
    sound( Trunc( Raport[ tasta+1-CodT[1] ] * Nota_Do[1] ) )
   end
   else
    if (tasta>=CodT[2])and(tasta<CodT[2]+10) then
    begin
     gotoxy(5*(tasta+1-CodT[2]),22);write(Nota[tasta+1-CodT[2]]);
     sound( Trunc( Raport[ tasta+1-CodT[2] ] * Nota_Do[2] ) )
    end
    else
       if (tasta>=CodT[3])and(tasta<CodT[3]+10) then
       begin
        gotoxy(5*(tasta+1-CodT[3]),20);write(Nota[tasta+1-CodT[3]]);
        sound( Trunc( Raport[ tasta+1-CodT[3] ] * Nota_Do[3] ) )
       end
        else
          if (tasta>=CodT[4])and(tasta<CodT[4]+10) then
          begin
            gotoxy(5*(tasta+1-CodT[4]),18);write(Nota[tasta+1-CodT[4]]);
            sound( Trunc( Raport[ tasta+1-CodT[4] ] * Nota_Do[4] ) )
          end
          else nosound;
 until tasta=129;
 SetIntVec($9,@KbdIntVec);
End.
```

**Program Testare_VESA;**

```
{ Program de testare a posibilitatilor de lucru a placii grafice in
standardul VESA. }
uses dos;
type tmoduri=array[1..256] of word;

var imod,vseg,x,y:word; cbank,c:longint; rg:registers;
   ntbanks:longint; opt:char;
   vesabuf:record sign:longint; vers:word; oem:pchar;
               capab:longint; list:^tmoduri;
               reserv:array[1..512] of byte end;
   vesamod:record attr:word; wina,winb:byte;
               gran,winsiz,sega,segb:word; pagfun:pointer;
               bytes,width,height:word;
               charw,charh,planes,bits,nbanks,model,sbank,
               nrimpg,reservb,rms,rfp,gms,gfs,bms,bfs:byte;
               reserv:array[1..512] of byte end;

function hexa(v:word):string;
const s:string[16]='0123456789abcdef';
function hexb(b:byte):string;
begin
    hexb:=s[b div 16+1]+s[b mod 16+1];
end;
begin
    hexa:=hexb(hi(v))+hexb(lo(v));
end;
```

```
procedure setbank(b:longint);
begin
    vseg:=$a000;
    if b<>cbank then with rg,vesamod do begin
      cbank:=b; ax:=$4f05; bx:=0;
      dx:=b*64 div gran; intr(16,rg);
      end;
end;

procedure putpixel(x,y:word; cul:longint);
var l:longint; m,z:word;
begin
    with rg,vesamod do case bits of
        4: begin
          l:=longint(bytes)*y+x div 8;
          port[$3ce]:=3; port[$3cf]:=0;
          port[$3ce]:=5; port[$3cf]:=2;
          port[$3ce]:=8; port[$3cf]:=128 shr (x and 7);
          setbank(l shr 16);
          z:=mem[vseg:word(l)]; mem[vseg:word(l)]:=cul;
          end;
        8: begin
          l:=longint(bytes)*y+x; setbank(l shr 16);
          mem[vseg:word(l)]:=cul;
          end;
        15,16: begin
             l:=longint(bytes)*y+x*2; setbank(l shr 16);
             memw[vseg:word(l)]:=cul;
             end;
        24: begin
            l:=longint(bytes)*y+x*3;
            z:=word(l); m:=l shr 16; setbank(m);
            if z<$fffe then move(cul,mem[vseg:z],3)
            else begin
                mem[vseg:z]:=lo(cul);
                if z=$ffff then setbank(m+1);
                mem[vseg:z+1]:=lo(cul shr 8);
                if z=$fffe then setbank(m+1);
                mem[vseg:z+2]:=cul shr 16;
                end;
           end;
        end;
end;

begin
    with rg, vesabuf, vesamod do begin
        ax:=$4f00; es:=seg(vesabuf); di:=ofs(vesabuf);
        sign:=$41534556; intr(16,rg);
        if al<>$4f then begin
          writeln('Standardul VESA nu e implementat');
          exit end;
        imod:=1;
        while list^[imod]<>$ffff do begin
            ax:=3; intr(16,rg); ax:=$4f01; cx:=list^[imod];
            es:=seg(vesamod); di:=ofs(vesamod);
            intr(16,rg);
            if attr and 16<>0 then begin
              writeln(oem,' VESA Versiune ',hi(vers),'.',lo(vers));
              writeln(hexa(list^[imod]),
              '     Rezolutie: ',width,' x ',height,
```

```
            '     Culori: ',longint(1) shl bits);
            write('Doriti testare (D/N)? '); readln(opt);
            end else opt:='N';
          if upcase(opt)='D' then begin
            ax:=$4f02; bx:=list^[imod];
            intr(16,rg); cbank:=-1;
            ntbanks:=longint(bytes)*height div gran div 1024;
            for x:=0 to ntbanks do begin
              setbank(x); mem[$a000:$ffff]:=0;
              fillchar(mem[$a000:0],$ffff,0);
              end;
            case bits of
               4,8: c:=15;
               15: c:=32767;
               16: c:=65535;
               24: c:=longint(1) shl 24-1;
               end;
            for x:=0 to width-1 do begin
              putpixel(x,0,c); putpixel(x,height-1,c);
              end;
            for y:=0 to height-1 do begin
              putpixel(0,y,c); putpixel(width-1,y,c);
              end;
            for x:=0 to 191 do for y:=0 to 191 do begin
              case bits of
                 4: c:=(y div 48)*4+x div 48;
                 8: c:=(y div 12)*4+x div 12;
                 15,16: c:=(y div 6)*(1 shl rfp)+x div 6;
                 24: c:=longint(x)*65536+y;
                 end;
              putpixel(x+4,y+4,c);
              end;
            readln;
            end;
          inc(imod);
          end;
        ax:=3; intr(16,rg);
        end;
end.
```

# Curiozităţi şi trucuri de programare

Pentru o cît mai completă prezentare a programării în C nu puteam evita prezentarea unor curiozităţi şi ale unor trucuri de programare C. Acelaşi lucru este valabil şi pentru limbajul Pascal dar este acesta este oarecum "ieşit din modă". Numărul foarte mare de astfel de "invenţii" a condus la organizarea încă din 1984 a unui concurs internaţional de programare numit foarte sugestiv *The International Obfuscated C Code Contest – IOCCC* adică *Concursul internaţional de programare ofuscată C* (încîlcită şi confuză). Participanţii la acest concurs oferă în fiecare an adevărate perle de programare C ce dovedesc, pe lîngă serioase cunoştinţe de C, aptitudinile extraordinare şi fiabilitatea compilatorului C. Multe din capodoperele acestui concurs au fost apoi înscripţionate pe tricouri sau pungi, spre deliciul fanilor programării C.

Această pasiune are totuşi şi o latură serioasă ce poate fi sesizată în programarea sub platformele (sistemele de operare) gen Unix. În aceste sisteme toate programele circulă nu numai sub forma de cod executabil ci şi în sursa originală C. Ascunderea unor informaţii despre programarea sistem de ochii celor "periculos" de curioşi este astfel dificilă. Dar iată că acest tip de programare "ofuscată" face acest lucru posibil ! Numai cei foarte pasionaţi îşi "prind urechile" în descifrarea unor astfel de programe intenţionat încîlcite. Altfel spus, secretul unor astfel de programe se ascunde chiar în rebusul din faţa ochilor cititorului.

Recomandăm acest capitol în special fanilor programării C şi celor foarte pasionaţi.

**// Un simplu "Hello world!" dar care arata o surprinzatoare interpretare a compilatorului C**

```
#include <stdio.h>
char a[]="Hello world!";
int i;
void main(void){
 for(i=0;a[i]!='\0';i++)
     putchar(i[a]);              // !! a[i] <=> *(a+i) <=> *(i+a) <=> i[a] !!
}
```

**// Iata unde conduce folosirea tipului de date float:**
**// c este foarte diferit de w ?!**
**// Putem spune ca acesta este un bug al C-ului ?**

```
#include <stdio.h>
float a=12345679.,b=12345678.,
          c=a*a-b*b,
          u=a*a,v=b*b,w=u-v;
void main(){
printf("a=%f,b=%f\nc=%f,w=%f\n",a,b,c,w);
}
```

**// Iata si varianta "corecta" in care nu se produce nici o trunchiere:**

```
#include <stdio.h>
long double a=12345679.,b=12345678.,
          c=a*a-b*b,
          u=a*a,v=b*b,w=u-v;
void main(){
printf("a=%Lf,b=%Lf\nc=%Lf,w=%Lf\n",a,b,c,w);
}
```

**// Acest program este capabil sa-si duplice identic la "iesire" codul sursa C fara a efectua nici o**
**// citire de nicaieri. Are deci caracteristica unui virus, se auto-replica !**

```
#include <stdio.h>
char *s[]={
"#include <stdio.h>",
"char *s[]={",
"void main(void){",
"int i;char *ps;",
"puts(s[0]);puts(s[1]);",
"for(i=0;i<10;i++)",
" {putchar(34);for(ps=s[i];*ps;ps++)putchar(*ps);",
" putchar(34);putchar(',');putchar(10);}",
"putchar(34);for(ps=s[10];*ps;ps++)putchar(*ps);putchar(34);putchar(10);",
"putchar('}');putchar(';');putchar(10);",
"for(i=2;i<11;i++)puts(s[i]);putchar('}');"
};
void main(void){
int i;char *ps;
puts(s[0]);puts(s[1]);
for(i=0;i<10;i++)
 {putchar(34);for(ps=s[i];*ps;ps++)putchar(*ps);
 putchar(34);putchar(',');putchar(10);}
putchar(34);for(ps=s[10];*ps;ps++)putchar(*ps);putchar(34);putchar(10);
putchar('}');putchar(';');putchar(10);
for(i=2;i<11;i++)puts(s[i]);putchar('}');
}
```

**// Program C surpriza (ales dintre cele de la IOCCC)**
**// Ce face acest program intr-o singura linie ?**

```
int i;main(){for(;i["]<i;++i){--i;}"];read('-'-'-',i+++"hello, world!\n",'/'/'/'));}read(j,i,p){write(j/p+p,i---
j,i/i);}
```

**// Alt program C surpriza (ales dintre cele de la IOCCC)**
**// Ce face acest program intr-o singura linie ?**

```
main(v,c)char**c;{for(v[c++]="Hello, world!\n)";(!!c)[*c]&&(v--||--
c&&execlp(*c,*c,c[!!c]+!!c,!c));**c=!c)write(!!*c,*c,!!**c);}
```

**// Puteti "decripta" acest program C de trei linii ? Executia lui arata clar ce face, intrebarea este**
**// insa cum face ?!**

```
#define P(X)j=write(1,X,1)
#define C 39
int M[5000]={2},*u=M,N[5000],R=22,a[4],l[]={0,-1,C-1,-1},m[]={1,-C,-1,C},*b=N,
*d=N,c,e,f,g,i,j,k,s;main(){for(M[i=C*R-1]=24;f|d>=b;){c=M[g=i];i=e;for(s=f=0;
s<4;s++)if((k=m[s]+g)>=0&&k<C*R&&l[s]!=k%C&&(!M[k]||!j&&c>=16!=M[k]>=16))a[f++
]=s;if(f){f=M[e=m[s=a[rand()/(1+2147483647/f)]]+g];j=j<f?f:j;f+=c&-16*!j;M[g]=
c|1<<s;M[*d++=e]=f|1<<(s+2)%4;}else e=d>b++?b[-1]:e;}P(" ");for(s=C;--s;P("_")
)P(" ");for(;P("\n"),R--;P("|"))for(e=C;e--;P("_ "+(*u++/8)%2))P("| "+(*u/4)%2
);}
```

# Confruntare de opinii: Informatică versus Matematică

Deşi poate părea neobişnuit pentru o culegere de probleme, am ţinut totuşi să introducem acest capitol pentru "a-i pune în gardă" pe începătorii într-ale informaticii de capcana confruntărilor sterile, pro informatică sau contra matematicii.

E bine ca ei să afle că deşi *informatica* este studiată ca ştiinţă de sine stătătoare ea este totuşi oficial considerată şi clasificată ca o sub-disciplină a matematicii. Desigur, acest fapt zgîndăre orgoliul unor "informaticieni pur-sînge" care, neînţelegînd că aceste clasificări sînt pur formale, intră deseori în confruntări aprinse de opinii cu matematicienii conservatori pe tema apartenenţei teoriilor informatice la matematică. Aceste sterile discuţii în contradictoriu nu pot fi însă auzite în mediile cu adevărat ştiinţifice, acolo unde se întîlnesc cei mai pasionaţi şi mai profunzi cercetători ai ambelor discipline.

Putem rezuma opiniile contradictorii, pe care le-am auzit şi noi deseori, sub forma următoarelor două întrebări care formulează în două moduri distincte aceeaşi dilemă:

1. *Se bazează informatica în întregime pe matematică sau ea are o existenţă separată ?*
2. *Se poate "face" informatică fără să cunoşti matematică foarte bine ?*

Înainte de a oferi răspuns, vom lămuri mai întîi o altă confuzie ceea ce ne va permite să răspundem mai uşor la cele două întrebări: care este diferenţa dintre *informatică* şi *ştiinţa calculatoarelor (computer science)* ?

Se ştie că există în facultăţile de la noi din ţară două (chiar trei) secţii cu profil informatic: *secţia de informatică* la facultatea de ştiinţe, *secţia de calculatoare* la facultatea de inginerie şi, mai nou, *secţia de prelucrare electronică a informaţiei economice (informatică economică)* la facultatea de ştiinţe economice. Sînt aceste secţii esenţial diferite ?

Să vedem o opinie cu "greutate". Iată cuvintele academicianului Nicolae Teodorescu despre informatică (am pus în evidenţă prin litere îngroşate cuvintele ce ni s-au părut esenţiale): "*Calculatorul electronic are însă ca merit esenţial stimularea unui mod de gîndire care* ***aştepta de veacuri*** *un mijloc tehnic prodigios pentru a da minţii omeneşti putinţa hotărîtoare de a-l introduce în strategiile investigative de avangardă. Acesta este modul de* ***gîndire algoritmică*** *care permite sortarea, analiza şi prelucrarea unui număr mare de posibilităţi, precum şi alegerea celei sau celor mai potrivite care conduc la rezultatul sau rezultatele urmărite, în studiul unor procese complexe care trebuie să fie simplificate sau abandonate din lipsă de mijloace de cercetare. Pentru promovarea acestei gîndiri, calculatorul electronic nu era însă suficient el însuşi, ci avea nevoie de o serie de discipline ştiinţifice avînd ca bază* ***gîndirea algoritmică****. Astfel, în puţinii ani de la introducerea calculatorului electronic s-au format discipline constituind o nouă ramură a ştiinţei cu caractere mixte teoretice şi tehnice, numită la un moment* ***informatică*** *termen care a înlocuit pe cel iniţial de* ***ştiinţă a calculului*** *sau* ***ştiinţă a calculatoarelor (computer science)*** *, care avea un înţeles mai precis, dar în acelaşi timp mai restrîns.*"

Vedem că, dintre cei toţi termenii de specialitate ce se folosesc, cea mai largă accepţiune o are termenul de *informatică*. Ceilalţi termeni, cum sînt *ştiinţa calculatoarelor* şi *informatică economică*, nu fac decît să nuanţeze şi să particularizeze înţelesul iniţial mai general. *Ştiinţa calculatoarelor* abordează informatica de pe poziţii inginereşti, ea primind un aport subtanţial de la alte discipline inginereşti ca electronica, ştiinţa prelucrării semnalelor electrice sau ştiinţa telecomunicaţiilor. *Informatica economică* utilizează noţiuni cu caracter strict economic sau din domeniul ştiinţelor sociale. Putem deduce că toate aceste nuanţări şi specializări au apărut din necesitate, datorită impactului deosebit pe care utilizarea pe scară largă a calculatoarelor îl are asupra sectoarelor societăţii.

Dacă însă vom grupa disciplinele cu caracter informatic care se predau simultan la fiecare din aceste secţii diferite vom obţine lista disciplinelor de bază ale informaticii: *Bazele informaticii, Programare, Structuri de date şi algoritmi, Sisteme de operare, Baze de date.* Alte discipline, cum sînt *Arhitectura calculatoarelor, Reţele de calculatoare, Ingineria programării, Inteligenţa artificială, Programarea orientată obiect*, etc., sînt considerate a fi discipline de specialitate în domeniu. De altfel, datorită acestor diferenţieri şi specializări între secţii, absolvenţii secţiilor respective se vor numi *programatori*, *ingineri de sistem* sau *economişti-informaticieni*. Să recunoaştem că s-ar ajunge la o adevărată "babilonie" dacă nu numai matematicienii ci şi inginerii sau economiştii şi-ar disputa cu informaticienii "puri" întîietatea în domeniile informatice ce le revin !

Rămîne să răspundem la întrebarea iniţială (formulată în două variante): în ce măsură se poate face informatică fără matematică ? Privind lucrurile la fel de pragmatic ca şi mai sus, dacă privim informatica ca pe o meserie (cu sub-specializările ei) iar matematica tot ca pe o meserie, este evident că nu este necesar să cunoşti două meserii pentru a o profesa bine pe una dintre ele. Deci, poţi fi un bun programator, inginer de sistem sau economist-informatician fără să ai cunoştinţe serioase de matematică. Trebuie însă să spunem, spre dezamăgirea celor "leneşi", că este exclus să fi lipsit de cunoştinţe de matematică pentru că atunci nu ai avea cum să-ţi însuşeşti cunoştinţele minimale pe care le oferă disciplinele de bază ale informaticii înşirate mai sus. Aceste discipline de bază fac apel la

modele şi metode matematice considerate deja clasice şi care sînt privite ca şi cultură matematică indispensabilă oricărui specialist în domeniu. Cum s-a ajuns la acest fapt, cum de găseşti matematică în economie şi în inginerie, dar nu şi invers ?

Este marele atu al matematicii: capacitatea de extragere a esențialului şi capacitatea de abstractizare (adică, capacitatea de modelare matematică). De altfel, este cunoscut faptul că cunoştinţele matematice esențiale, indiferent de forma în care ele sînt formalizate sau simbolizate, sînt aceleaşi pentru orice civilizație terestră. Sau extraterestră ! Se ştie că mesajele de pe sondele spațiale americane, ce au părăsit deja sistemul nostru solar, destinate unor posibile civilizații extraterestre sînt "scrise" în limbaj matematic. Să nu ne mai mirăm atunci că "fără matematică nu se poate !".

Ca să nu creadă cineva că facem pledoarie pentru matematică, aici într-o lucrare de informatică, vă facem cunoscut că, din contră, în cartea sa *Vîrsta de aur a matematicii,* care prezintă în 11 capitole cele mai mari realizări ale matematicii din ultimii 50 de ani, profesorul şi cercetătorul Keith Devlin de la universitățile Stanford şi Pittsburgh a introdus un capitol cu titlul *Eficiența algoritmilor* şi în alte cinci capitole arată rolul important pe care l-a avut folosirea calculatorului în creşterea eficienței şi validării cercetării pur matematice. Adică, şase din unsprezece capitole cer pentru a fi înțelese bine nu numai cunoştințe de matematcă ci şi de informatică. Iar unul din cele cinci capitole, *Problema celor patru culori*, accentuează rolul esențial (indispensabil) al programării în demonstrarea ***cu ajutorul calculatorului*** a uneia din cele mai celebre probleme de matematică. Această demonstrație a creat o "breşă" serioasă în gîndirea matematicienilor care au fost nevoiți să ia foarte în serios "concurența" pe care calculatorul (bine "dirijat" de programatori) a început să le-o facă. Iată chiar cuvintele profesorului de matematică Keith Devlin scrise în încheierea capitolului respectiv (ce explică modul în care s-a făcut demonstrația cu calculatorul): *"Matematica nu va mai fi niciodată aceeaşi."* !

Încheiem cu convingerea că, cei care au parcurs cu interes această culegere, inclusiv acest capitol, nu vor mai putea fi tentați de controverse "uşoare" *informatică versus matematică*. Credem că s-a putut vedea cum, cei care "sînt deasupra" acestor discuții sterile, au sesizat cu înțelepciune că matematica - "mama informaticii" - se îmbogățeşte acum din plin prin intermediul informaticii, "punîndu-le astfel pe picior de egalitate" cele două discipline.

Noi le urăm tuturor celor studioşi să-şi concentreze toată energia pasiunii lor pentru învățarea şi stăpînirea cu măiestrie a "artei programării". Ea poate fi considerată ca fiind prima treaptă importantă spre orizontul către care tinde ştiința informaticii.

## Bibliografie, adrese şi locaţii de interes pe Internet

Internetul e foarte mare, stufos şi, de multe ori, labirintic. Tocmai de aceea, ne-am gîndit să venim în ajutorul celor foarte pasionaţi de informatică şi de matematica aplicată în informatică. Oferim în continuare doar cîteva adrese pe care şi noi le-am utilizat cu succes. Fiecare din aceste site-uri conţine la rîndul lui liste de adrese şi legături (links) către alte site-uri cu subiecte asemănătoare. Iată, aveţi la dispoziţie "un capăt al ghemului" !

- www-groups.dcs.st-and.ac.uk/~history/ - conţine multe pagini interesante despre istoria descoperirilor în matematică, utile celor care doresc să afle cum se face cu adevărat descoperiri în matematică şi cum s-a ajuns la necesitatea apariţiei calculatoarelor
- www.mathpages.com/KsBrown/ - conţine o colecţie impresionantă de informaţii, idei şi descoperiri de ultimă oră din matematică şi informatică
- www.mathsoft.com/asolve/ - conţine o listă substanţială de probleme de matematică (şi nu numai) care îşi aşteaptă încă rezolvarea, multe dintre ele putînd fi abordate cu ajutorul calculatorului
- www.ee.Surrey.ac.uk/Personal/R.Knott/Fibonacci/fib.html - este o "portiţă" de intrare în domeniul fascinant al numerelor lui Fibonacci, cu multiple corelaţii matematice şi informatice
- mans.cee.hw.ac.uk/ctl.html *Computer Teaching and Learning Resources* - numele site-ului spune totul
- www.k12tlc.net/Penrose/ *K-12 Teaching & Learning Center* - noi am ales pagina care prezintă biografia lui Sir Roger Penrose, dar aveţi încă multe altele la dispoziţie
- www.ioccc.org *The International Obfuscated C Code Contest (IOCCC)* – Concursul internaţional de programare C ofuscată (încîlcită şi intenţionat confuză)

Suplimentar, tot pentru cei foarte pasionaţi de matematică, informatică, de legătura dintre ele şi nu numai, oferim o selecţie minimală de cărţi şi articole care au constituit, direct sau indirect, o sursă de inspiraţie în scrierea acestei culegeri:


- *Turbo Pascal 6.0. Ghid de utilizare*, Microinformatica, Cluj-Napoca, 1992
- **Bălănescu T. …,** *Limbajul Turbo Pascal*, Editura tehnică, Bucureşti, 1992
- **Grigore Albeanu,** *Programarea în Pascal şi Turbo Pascal. Culegere de probleme*, Editura tehnică, Bucureşti, 1994
- **Tudor Sorin,** *Tehnici de programare,* Editura L&S Infomat, Bucureşti, 1998
- *Manual de programare C,* (după **Kernigham** şi **Ritchie**) Microinformatica, Cluj-Napoca, 1986
- **Muşlea I.**, *Programarea în C,* Microinformatica, Cluj-Napoca, 1992
- **Roger Penrose,** *Mintea noastră…cea de toate zilele*, (titlul original: *Emperor's mind*), Editura tehnică, Bucureşti, 2001
- **Roger Penrose,** *Incertitudinile raţiunii. Umbrele minţii*, (titlul original: *Shadows of the mind*), Editura tehnică, Bucureşti, 2000
- **Keith Devlin,** *Vîrsta de aur a matematicii*, (titlul original: *Matemathics: The New Golden Age*), Editura Thetha, Bucureşti, 2000
- **Solomon Marcus,** *Gîndirea algoritmică*, Editura tehnică, Bucureşti, 1982
- **L. Livovschi, H. Georgescu**, *Bazele informaticii*, Editura didactică şi pedagogică, Bucureşti, 1981